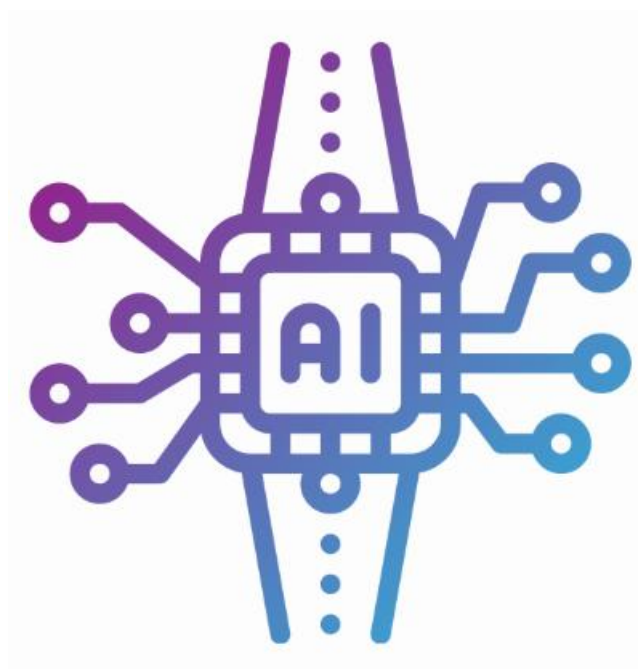


Yuriy Fedkovych Chernivtsi National University
Faculty of Mathematics and Computer Science

H.Melnyk, V.Melnyk

Artificial Intelligence Systems

Textbook



Chernivtsi, 2025

Published by the decision of the Editorial and Publishing Council
of Yuriy Fedkovych Chernivtsi National University

A-912 Artificial Intelligence Systems: A Textbook / Compiled by V.
Melnyk, H. Melnyk – Chernivtsi: CNU, 2025. – 420 p.

This textbook provides theoretical foundations and
methodological guidelines for laboratory and practical tasks in
the course "*Artificial Intelligence Systems*."

Recommended for students of the Faculty of Mathematics and
Informatics.

Content

Introduction to Artificial Intelligence (AI)	4
Introduction to AI. Practical Tasks	22
AI Ethics, Security, and Confidence	27
AI Ethics. Practical Task	34
Search Algorithms in AI	37
Search Algorithms in AI. Practical Tasks	87
Optimization in AI	94
Optimization. Practical Tasks	116
Machine Learning in AI	123
Machine Learning. Practical Tasks	175
Neural Networks in AI	188
Neural Networks. Practical Tasks	329
Natural Language Processing in AI	336
Natural Language Processing (NLP). Practical Tasks	352
Computer Vision in AI	354
Computer Vision in AI. Practical Tasks	388
Genetic Algorithms in AI	395
Genetic Algorithm. Practical Tasks	416
References	420

Foreword

The discipline of *Artificial Intelligence Systems* occupies a central place in modern computer science and engineering. In today's world, where intelligent technologies are no longer confined to research laboratories but are widely integrated into business, healthcare, education, transportation, and even our daily routines, the need for systematic and deep understanding of AI has become essential.

This textbook is designed to provide students with a comprehensive overview of the fundamental concepts, architectures, and applications of artificial intelligence systems. It combines theoretical foundations with practical examples, ensuring that learners not only acquire knowledge of the principles behind AI but also develop the skills necessary to apply these concepts in real-world scenarios.

The material is structured to gradually guide students from the basic notions of intelligent systems, through classic methods such as search algorithms, knowledge representation, and reasoning, to more advanced topics including machine learning, neural networks, and intelligent decision-making. Special attention is given to the ethical and societal implications of AI, highlighting the responsibility that comes with creating and deploying intelligent technologies.

This textbook is intended for students of computer science, applied mathematics, and related fields, as well as for educators, researchers, and practitioners interested in understanding how artificial intelligence systems are built and how they shape the future of technology.

It is our hope that this material will serve not only as a source of knowledge but also as an inspiration to further exploration, innovation, and responsible use of AI in academic and professional contexts.

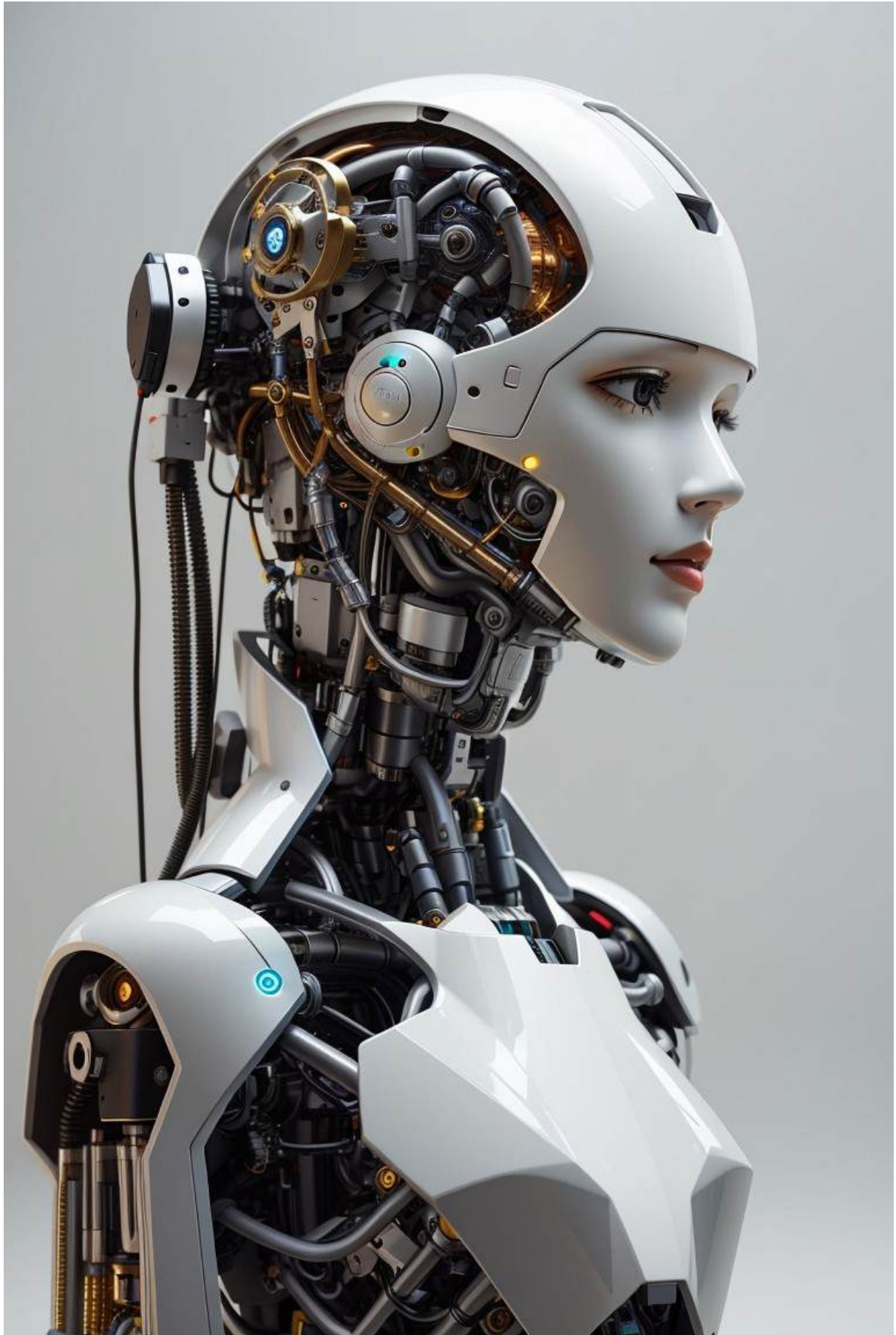
Topic: Introduction to Artificial Intelligence (AI)

Duration: 2 hours

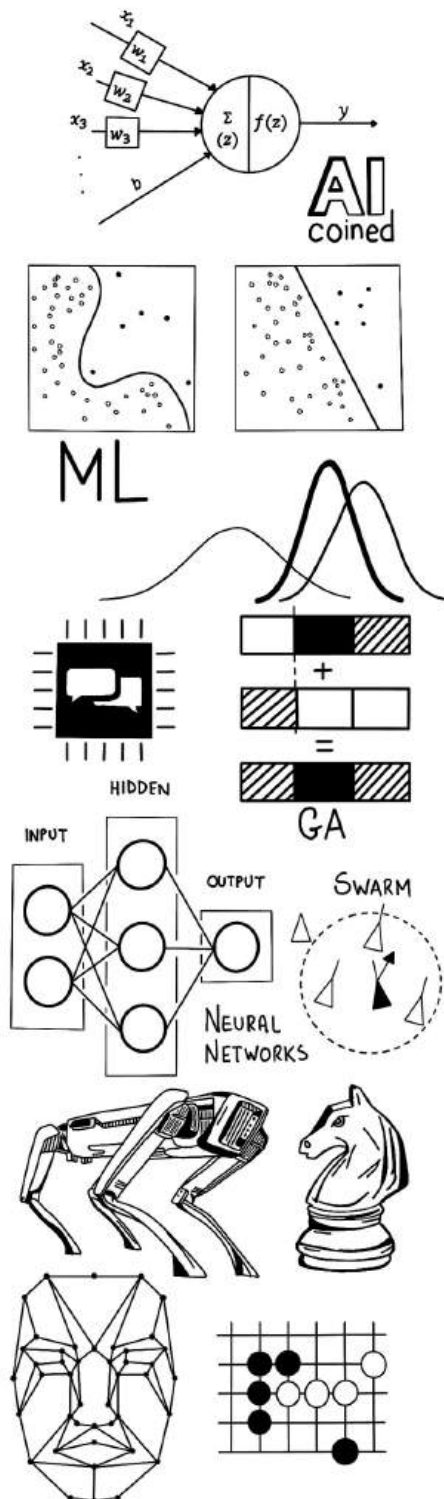
Agenda:

1. What is Artificial Intelligence?
2. Types of Artificial Intelligence. AI Algorithms
3. AI Techniques and Approaches
4. The AI Development Process
5. Ethical Considerations in AI
6. Real-world Applications of AI
7. AI's Impact on Society and the Future
8. Q&A and Wrap-up

Made by Leonardo.AI



Looking at past developments highlights the importance of understanding the fundamentals of AI; algorithms from decades ago are critical in many modern AI implementations. Our session starts with fundamental algorithms that help build the intuition of problem-solving and gradually moves to more interesting and modern approaches.



rhurbans.com

1950s

- The term "artificial intelligence" is coined.
- Concept of artificial neural network is introduced.
- Model of the Perceptron is invented.
- LISP programming language is invented.

1960s

- ML models for prediction are introduced.
- Unimate robot works on a car assembly line.
- Shakey the robot has natural movement and problem-solving abilities.
- Paper highlighting the flaws of Perceptrons creates doubt about the concept.

1970s

- BKG wins at backgammon (with luck).
- Evolutionary algorithms are popularized.
- Freddy the robot is able to use visual perception.
- Prolog programming language is invented.

1980s

- LISP machines for expert systems.
- Hope for neural networks via the introduction of backpropagation.
- Swarm intelligence is popularized.

1990s

- TD-Gammon shows the power of reinforcement learning.
- Experiments with autonomous cars.
- IBM's Deep Blue becomes a chess champion.
- Rise of internet bots and search.

2000s

- Game of checkers is solved.
- Face recognition with neural networks.
- IBM's Watson wins at Jeopardy.
- XBox Kinect's advanced motion detection.
- Smart voice assistants by tech giants.
- Google's Alpha Go becomes a Go champion.
- AI-specific hardware and IoT devices.
- Tumor detection better than doctors.
- Self-driving cars.

Intelligence is a mystery. Intelligence is a concept that has no agreed upon definition. Philosophers, psychologists, scientists and engineers all have different opinions about what it is and how it emerges. We see intelligence in nature around us such as groups of living creatures working together, and we see intelligence in the way that we think and behave. In general, things that are autonomous yet adaptive are considered to be intelligent.

1. What is Artificial Intelligence?

Artificial Intelligence (AI) is the simulation of human intelligence in machines that are programmed to think and act like humans. The primary objective of AI is to enable machines to perform tasks that typically require human intelligence, such as problem-solving, decision-making, learning, and language understanding.

AI aims to create intelligent agents that can perceive their environment, reason about it, and take actions to achieve specific goals. These intelligent agents can learn from their experiences, adapt to new situations, and improve their performance over time.

Slides for [Introduction to Artificial Intelligence](#)

The idea of AI dates back to antiquity, but the modern development of AI as a field began in the mid-20th century. [Key milestones](#) in AI development.

AI has a vast scope and is being applied in various industries and fields, revolutionizing how we live, work, and interact. Some notable areas of AI application include:

- Healthcare: AI is used for medical diagnosis, drug discovery, personalized treatment plans, and disease prediction.
- Finance: AI powers algorithmic trading, fraud detection, credit risk assessment, and customer service chatbots.
- Transportation: AI is utilized in autonomous vehicles, traffic optimization, and predictive maintenance in aviation and railways.
- Retail: AI enables personalized product recommendations, inventory management, and customer service chatbots.
- Entertainment: AI is used in content recommendation systems for movies, music, and social media platforms.
- Education: AI enhances adaptive learning platforms, intelligent tutoring systems, and automated grading.
- Robotics: AI-driven robots are employed in manufacturing, logistics, and healthcare for tasks automation.

AI's impact continues to expand, and its potential to transform various industries is immense. As AI technologies advance, it becomes essential to understand its capabilities, benefits, and potential challenges, making it a significant topic for exploration in introductory AI courses.

2. Types of Artificial Intelligence. AI Algorithms

Traditionally scientists define two types of Artificial Intelligence: Narrow (Weak) AI and General (Strong) AI.

- Narrow AI (Weak AI) refers to AI systems that are designed and trained for a specific task or a narrow set of tasks. These systems excel at performing one task with high proficiency but lack the ability to generalize or perform tasks outside their specific domain. Examples of narrow AI include virtual assistants like Siri,

recommendation systems, image recognition algorithms, and language translation tools.

- General AI (Strong AI), also known as Artificial General Intelligence (AGI), refers to AI systems that possess human-like intelligence and have the ability to understand, learn, and apply knowledge across a wide range of tasks. Unlike narrow AI, which is specialized, general AI would be capable of reasoning, self-awareness, and learning from diverse experiences. Achieving General AI is a long-term goal of AI research, but it has not yet been fully realized.

Applied AI refers to the practical implementation of AI technologies to solve specific real-world problems. It involves using existing AI algorithms, models, and tools to address specific tasks or challenges in various domains. Applied AI focuses on creating practical solutions and often involves data processing, model training, and deploying AI systems in real-world applications. Examples of applied AI include AI-powered chatbots, fraud detection systems, medical diagnosis tools, and self-driving cars.

Research-oriented AI involves the development of new AI algorithms, techniques, and models through theoretical research. Researchers in this domain focus on pushing the boundaries of AI by exploring new approaches and methods to improve AI capabilities. Research-oriented AI contributes to advancements in machine learning, natural language processing, computer vision, and robotics. It plays a crucial role in driving innovation and enabling the development of more sophisticated AI applications.

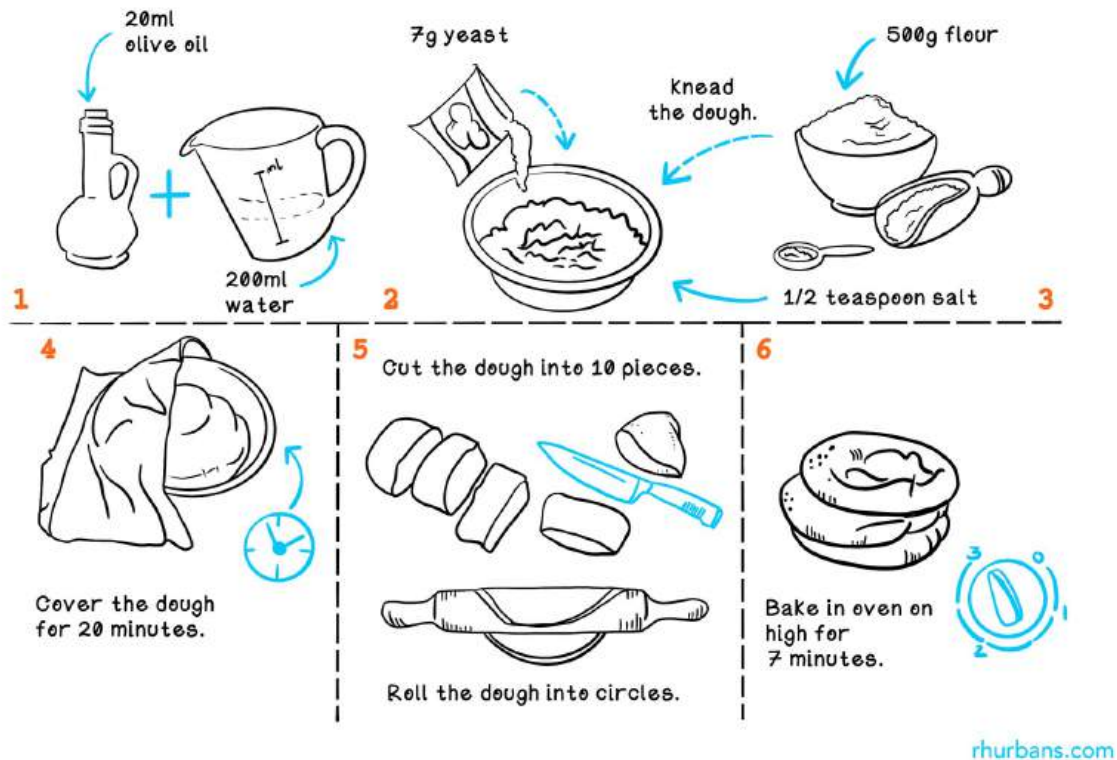
Real-World Examples of AI Applications in Daily Life

These examples demonstrate how AI is integrated into our daily lives, making it a powerful and pervasive technology with far-reaching applications in diverse domains. Understanding the types of AI and its real-world implementations provides valuable insights into its potential and impact on society and various industries.

Algorithms are like recipes

Algorithms are like a pita bread recipe. There's a problem being solved (making good pita bread), ingredients required (pieces of input), a sequence of steps to follow (recipe instructions), and the resulting output, in this case, pita bread of a certain quality.

Pita bread algorithm



Algorithms can provide the same output given a specific input, always - these are called deterministic algorithms; or they can result in a different output when the steps involve an element of randomness or uncertainty - these are called stochastic algorithms.

The algorithm for a $\max(x, y)$ function might involve comparing two numbers to find the maximum. Inputs are x and y , and the algorithm finds the largest - somewhat easy to intuit how it works.

The algorithm for Photoshop to cut a background might involve iterating through all the pixels in an image and finding edges or areas of contrast - there's a set of well known techniques for this.

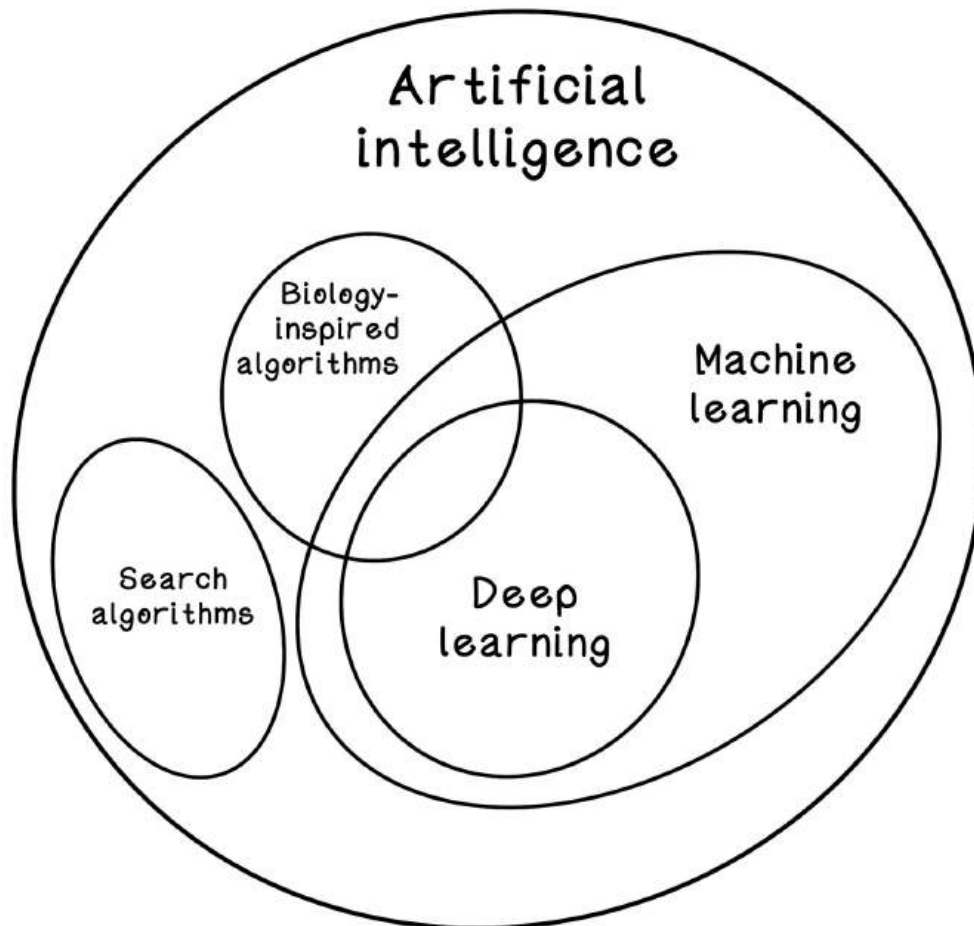
The algorithm for Netflix might involve comparing the category of videos you've watched with the categories of all other movies out there to provide you with recommendations...or, it might compare your demographic data with other user's data, and provide you with recommendations based on movies they have watched - these algorithms' outputs are more difficult to estimate.

You might have heard of the joke: An algorithm is a word used by programmers when they don't want to explain what they did.

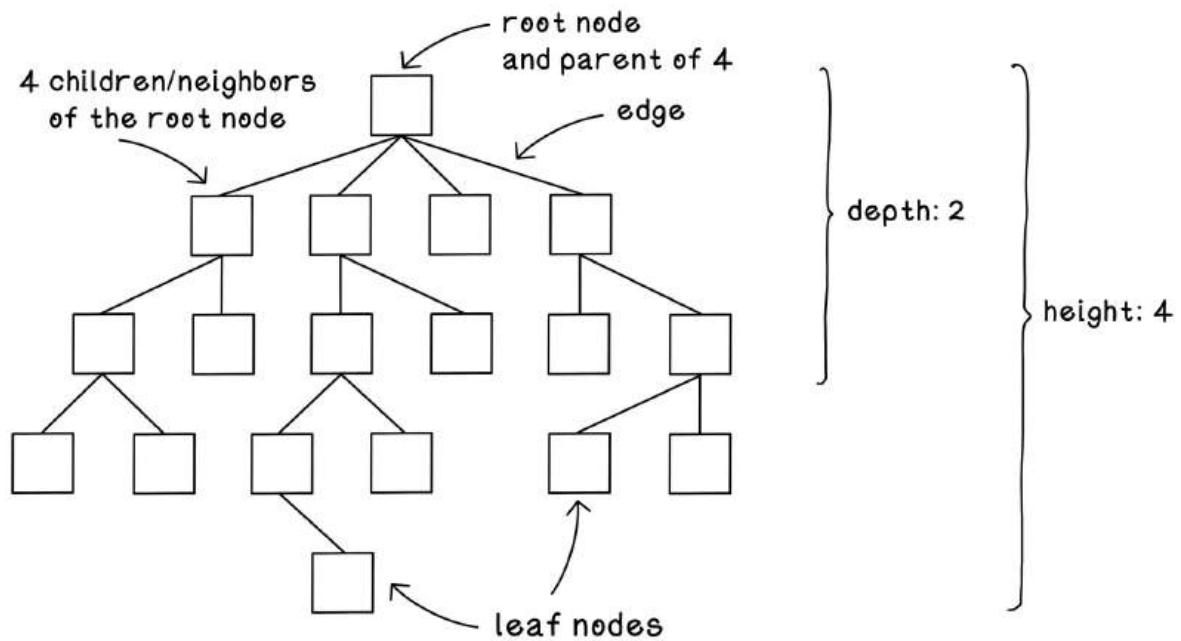
Deep learning models are solving some of the toughest problems today, but they're often mysterious in their workings. The future calls for algorithms to be more transparent, understandable, and accessible to people of varied skillsets - a mammoth of a challenge for us all!

AI algorithm families

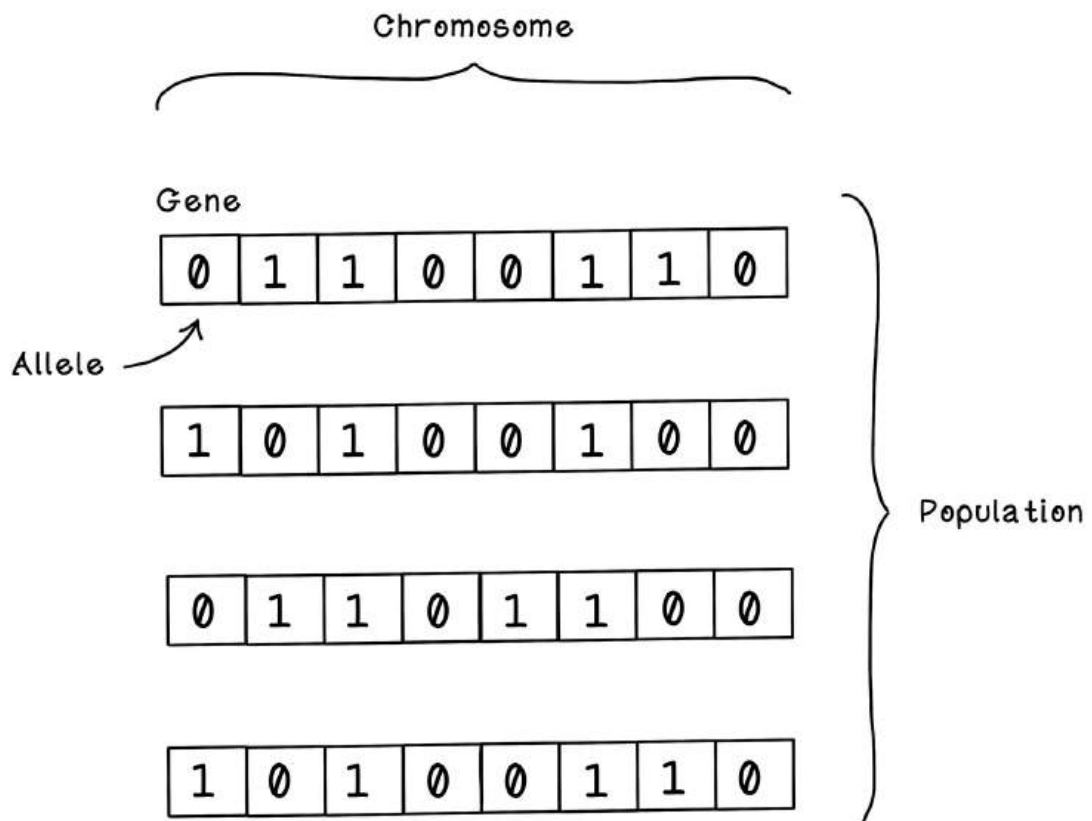
Different families of algorithms solve different problems. We don't necessarily need to be experts in the details of each one, but having a grasp on what problems they can solve, and how they generally work, equips us with more tools when making decisions.



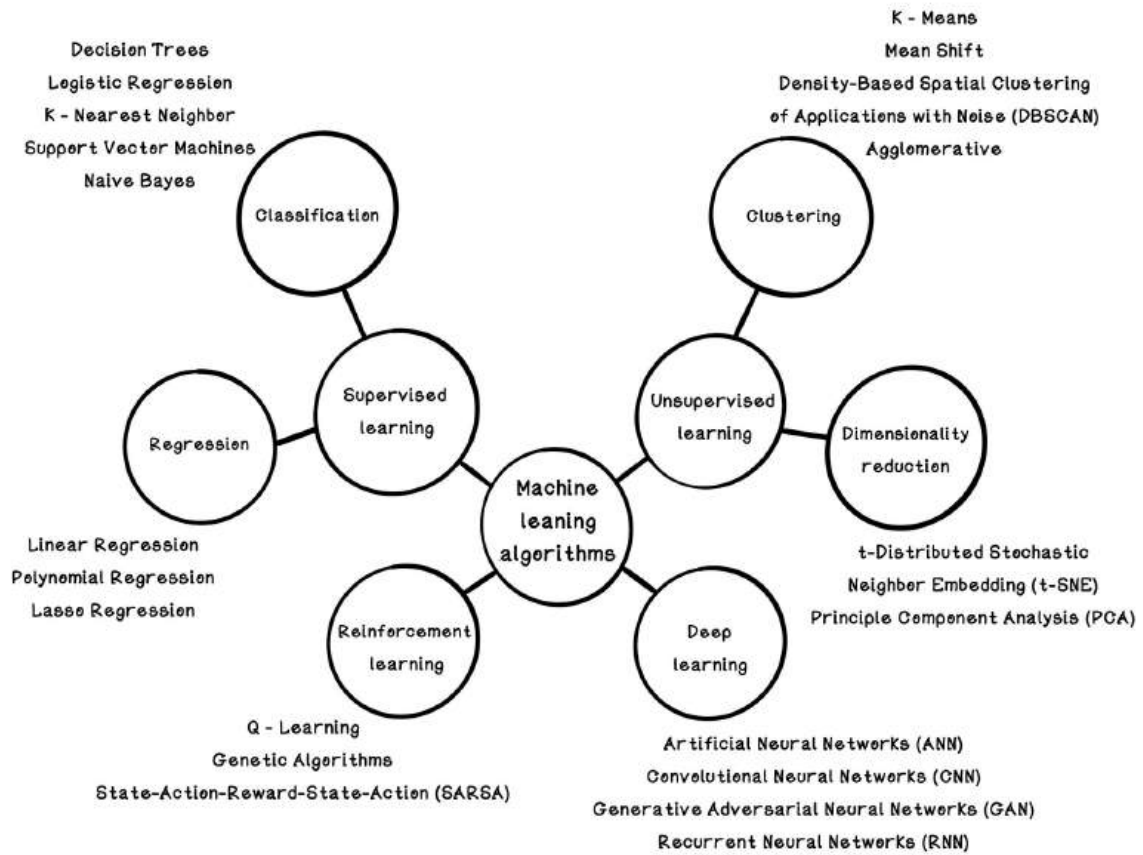
Traditional search algorithms are useful where several actions are required to achieve a goal, like finding a path through a maze. These algorithms evaluate possible states and attempt to find an optimal path. Typically, we have too many possible solutions to brute-force.



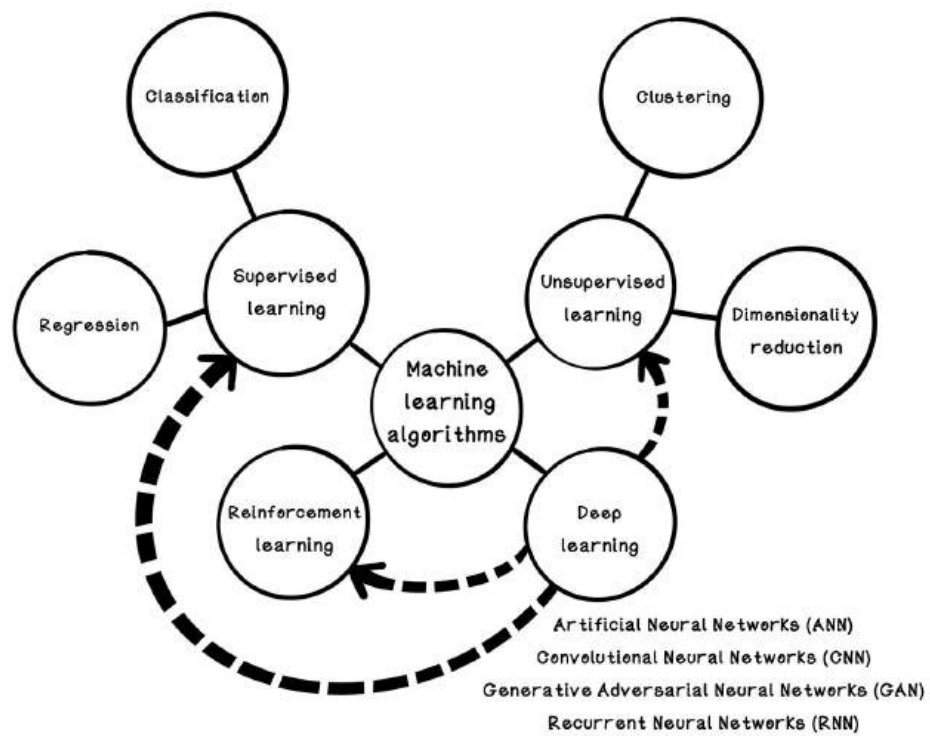
Biology-inspired algorithms are wondrous things happening all the time. The cooperation of ants in gathering food, the flocking of birds, estimating how brains work, and the evolution of organisms to produce strong offspring. These have inspired algorithms that are useful in AI.



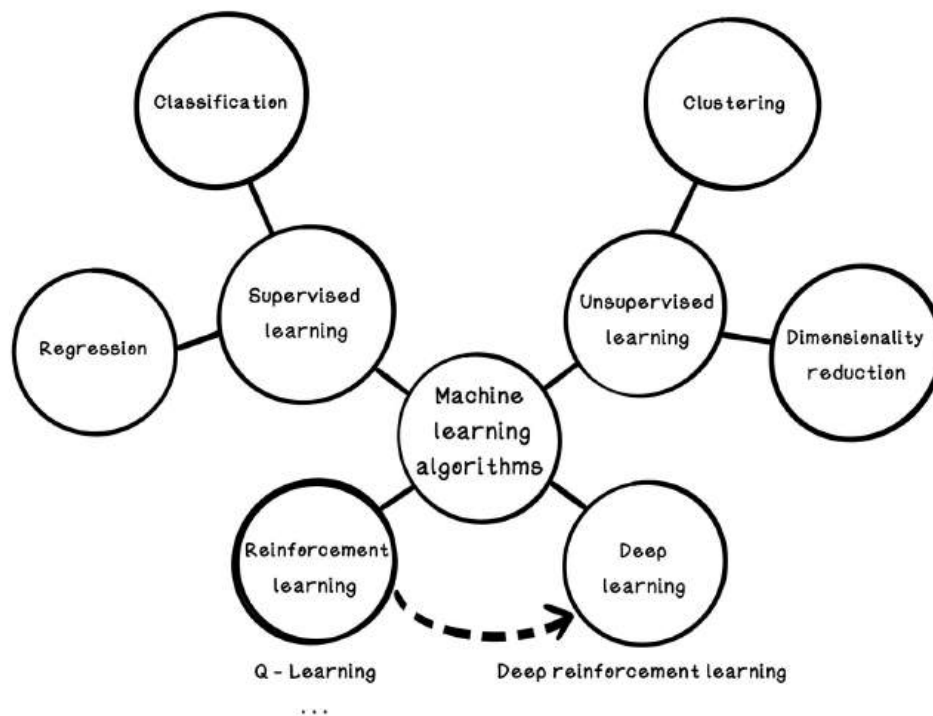
Traditional machine learning algorithms leverages statistics to training models to learn from data. The umbrella of machine learning has a variety of algorithms that can be harnessed to improve understanding of relationships in data, to make predictions and decisions.



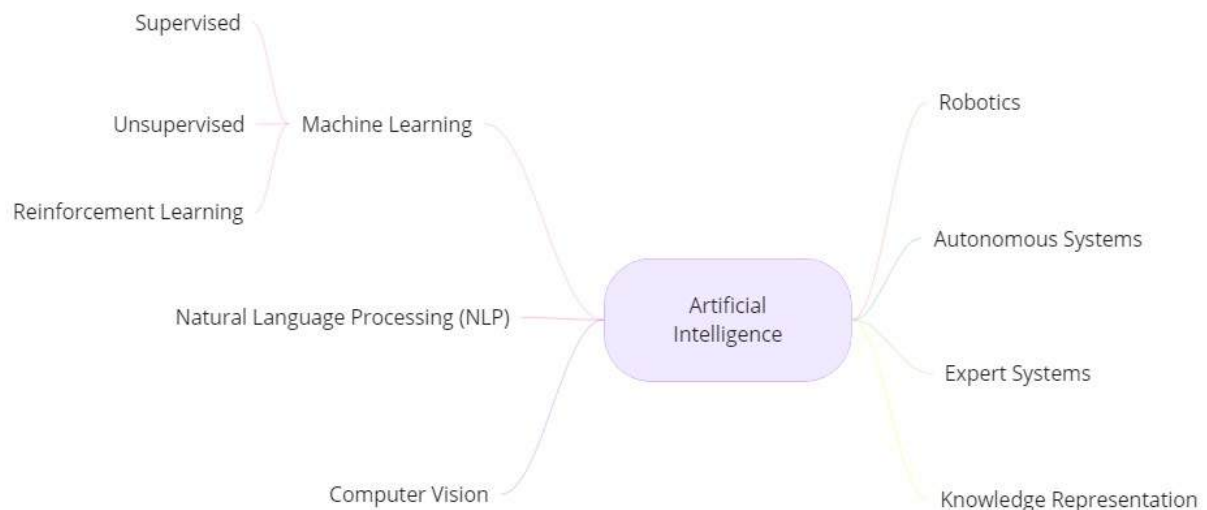
Deep learning algorithms are a broader family of approaches and algorithms that are used to achieve narrow intelligence and strive toward general intelligence. It attempts to solve general problems like vision, speech, and reasoning. It often leverages artificial neural networks.



Reinforcement learning algorithms are based on behavioural psychology and use feedback from actions performed to learn what sequences are more beneficial. Reinforcement learning is useful when you know what the goal is but don't know what actions are reasonable to achieve it.



3. AI Techniques and Approaches



In this item, we will explore the diverse world of Artificial Intelligence and the techniques used to enable machines to perform tasks similar to human intelligence. AI techniques have revolutionized various industries and are driving innovation and advancements in technology.

1. Machine Learning is a fundamental subset of AI that empowers machines to learn from data and improve their performance over time. There are three main types of Machine Learning:

- *Supervised Learning:* In this approach, the algorithm is trained on labeled data, where the input-output pairs are provided. It learns to map inputs to desired outputs and can be used for tasks like image classification and speech recognition.
- *Unsupervised Learning:* Unsupervised Learning deals with unlabeled data, and the algorithm aims to find patterns or structures within the data. It is useful for clustering and dimensionality reduction.
- *Reinforcement Learning:* This technique involves an agent interacting with an environment to achieve a goal. The agent receives rewards or penalties based on its actions, encouraging it to learn optimal strategies. Reinforcement Learning is applied in areas like game playing and robotics.

2. Natural Language Processing (NLP) is a branch of AI that focuses on enabling machines to understand, interpret, and generate human language. NLP plays a crucial role in various applications, including:

- *Chatbots and Virtual Assistants:* NLP powers conversational agents that can interact with users and respond to natural language queries.
- *Sentiment Analysis:* NLP helps determine the sentiment or emotion expressed in textual data, valuable for brand reputation management and market analysis.
- *Language Translation:* NLP algorithms facilitate the translation of text from one language to another, breaking down language barriers.

3. Computer Vision is an AI technique that enables machines to interpret and understand visual information from images or videos. Some key applications of Computer Vision include:

- *Object Detection:* Machines can identify and locate objects within images or videos, essential in security systems and autonomous vehicles.
- *Facial Recognition:* Computer Vision enables machines to recognize and verify individuals based on facial features, widely used in security and authentication.
- *Autonomous Vehicles:* Computer Vision plays a critical role in enabling self-driving cars to perceive and navigate their surroundings.

4. Robotics is an exciting field of AI that focuses on designing and building intelligent machines capable of interacting with the physical world. Key areas of AI in robotics include:

- *Autonomous Robots:* AI-powered robots can operate without human intervention, performing tasks in challenging environments or hazardous situations.
- *Industrial Automation:* Robotics with AI enables automation in manufacturing, streamlining production processes and improving efficiency.
- *Social Robots:* AI-driven social robots can interact with humans in social settings, assisting with companionship, therapy, or education.

Certainly! Let's add information about Expert Systems and Knowledge Representation in AI to the session on "AI Techniques and Approaches."

5. Expert Systems are a class of AI programs designed to emulate the decision-making capabilities of human experts in specific domains. They use knowledge representation techniques to store and organize expert knowledge, which is then used to solve problems or provide recommendations.

- Expert Systems are rule-based systems that use a knowledge base of facts and rules to arrive at conclusions or solutions. These rules are derived from human experts who possess expertise in a particular field.

- The knowledge base contains both factual information about the domain and a set of rules that guide the system's reasoning process.
- Expert Systems are particularly useful in domains where human expertise is crucial but scarce or costly, such as medical diagnosis, financial analysis, and troubleshooting complex technical issues.

6. Knowledge Representation is a fundamental aspect of AI, focused on representing and organizing knowledge in a way that machines can understand and use. Various formalisms are used for knowledge representation, including:

- *Rules:* Representing knowledge in the form of IF-THEN rules allows the system to make inferences based on given conditions.
- *Frames:* Frames are structures that organize knowledge hierarchically, capturing the properties and relationships of objects or concepts.
- *Ontologies:* Ontologies provide a formal description of concepts, relationships, and properties within a domain, facilitating knowledge sharing and reasoning.
- *Semantic Networks:* These graphical representations use nodes to represent concepts and edges to represent relationships between them.
- *Production Systems:* Production systems consist of a set of production rules, working as a knowledge representation mechanism in Expert Systems.

Expert Systems and Knowledge Representation play a vital role in AI, enabling machines to leverage human expertise and reasoning capabilities. These systems are powerful tools that have demonstrated valuable applications in various fields. As AI continues to advance, Expert Systems will remain an essential component in the quest to solve complex problems and improve decision-making processes.

AI techniques and approaches have a profound impact on various industries, transforming the way we live and work. From personalized recommendations to medical diagnosis and autonomous vehicles, AI is changing the landscape of technology and decision-making. We have only scratched the surface of what AI can achieve, and its potential continues to expand.

4. The AI Development Process

In this item, we will explore [the step-by-step approach](#) to developing Artificial Intelligence solutions, from data collection and preprocessing to model deployment and ongoing maintenance.

1. Data Collection and Preprocessing

- Data is the foundation of AI development. This step involves gathering relevant data from various sources to create a comprehensive dataset.
- Data preprocessing is essential to clean, transform, and prepare the data for training. It includes tasks like handling missing values, normalization, and feature engineering.
- Emphasize the significance of high-quality data as it directly influences the model's accuracy and performance.

2. Model Development and Training

- Model development involves selecting an appropriate AI algorithm or architecture based on the problem's nature and data characteristics.
- Explain the training process, where the model learns from the data and adjusts its parameters to optimize performance.

- Introduce popular AI models like Neural Networks, Decision Trees, and Support Vector Machines.

3. Model Evaluation and Fine-Tuning

- Model evaluation is crucial to assess the performance of the trained model. Metrics like accuracy, precision, recall, and F1-score are used to evaluate the model's effectiveness.
- Discuss the concept of overfitting and underfitting and the importance of fine-tuning model hyperparameters to achieve better generalization.
- Highlight the iterative nature of model evaluation and fine-tuning to continually improve performance.

4. Deployment and Ongoing Maintenance

- Deployment involves integrating the trained AI model into the intended application or system to make predictions or decisions in real-time.
- Discuss the challenges of model deployment, including scalability, latency, and system integration.
- Address the importance of ongoing maintenance to monitor the model's performance, update it with new data, and adapt to changing conditions.

The AI development process is a systematic and iterative approach that enables the creation of powerful AI solutions. From data collection and preprocessing to model development, evaluation, and deployment, each step is critical in achieving successful AI applications. Remember that AI development is an evolving journey, and continuous improvement and maintenance are essential to harness the full potential of AI technologies.

5. Ethical Considerations in AI

Our next step is to delve into the critical topic of ethics in Artificial Intelligence and explore the ethical challenges that come with the development and deployment of AI technologies. As AI continues to advance, it is essential to address ethical considerations to ensure responsible and beneficial use. Let's begin!

Artificial Intelligence Ethics refers to the study of moral principles and values concerning the development, deployment, and impact of AI systems on individuals, society, and the environment. As AI becomes more pervasive, it is crucial to consider its ethical implications to avoid harmful consequences and promote fairness, accountability, and transparency in AI applications.

One of the key ethical challenges in AI is *bias*. AI systems may unintentionally reflect and perpetuate existing societal biases present in the training data. This can lead to *unfair outcomes* for certain individuals or groups. Promoting fairness in AI algorithms ensures that the system treats all individuals equitably and without discrimination. However, achieving fairness in AI can be challenging, and it requires ongoing efforts to mitigate biases and ensure equitable outcomes.

Another critical ethical concern in AI revolves around *data privacy* and *protection*. AI systems often rely on vast amounts of data to train and make accurate predictions. Ensuring the privacy and security of user data is crucial. We must consider the ethical implications of data collection, storage, and usage, and obtain informed consent from users when using their data for AI training. Regulations like *GDPR* (General Data Protection Regulation) provide

guidelines for data privacy in AI systems and emphasize the importance of safeguarding user data.

Accountability is a vital aspect of AI ethics, particularly in critical applications like healthcare and autonomous vehicles. AI decision-making processes need to be accountable, and we must be able to trace how decisions are made. However, complex AI models can be challenging to interpret and explain, leading to concerns about transparency. Efforts are being made to develop "Explainable AI" to increase transparency and trust in AI systems.

Ethical considerations in AI are paramount to building responsible and trustworthy AI technologies. As developers, researchers, and users of AI, we have a responsibility to navigate the ethical challenges and make conscious decisions to ensure that AI benefits humanity while minimizing potential harm. By addressing bias, promoting fairness, protecting privacy, and increasing transparency, we can shape a future where AI serves as a force for positive change.

6. Real-world Applications of AI (Duration: 20 minutes)

- a. AI in healthcare and medical diagnosis
- b. AI in finance and fraud detection
- c. AI in recommendation systems and personalized advertising
- d. AI in autonomous vehicles and smart homes

7. AI's Impact on Society and the Future

AI has rapidly become an integral part of our lives, revolutionizing industries and shaping the way we interact with technology. In this session, we will discuss the transformative effects of AI, its implications on the workforce, ethical considerations, and the potential it holds for shaping our future.

Artificial Intelligence has *transformed* the way we live, work, and interact with the world around us. From voice-activated virtual assistants to personalized content recommendations, AI has become an integral part of our daily lives. Consider how AI enables:

- Enhanced customer experiences through chatbots and virtual customer support.
- Improved healthcare diagnostics and treatment plans through AI-powered medical imaging and data analysis.
- Increased efficiency in logistics and supply chain management through AI-driven optimization algorithms.
- Enhanced cybersecurity with AI systems detecting and responding to threats in real-time.

AI's rise has sparked debates about its impact on the *job market* and workforce. While AI automation may replace some repetitive tasks, it also creates opportunities for workforce reskilling and job creation in emerging fields:

- The importance of continuous learning and upskilling to adapt to the changing job landscape.
- The potential for AI to augment human capabilities, leading to a more productive and collaborative workforce.

- The need for policies and initiatives to support workers transitioning to AI-powered environments.

As AI becomes more *pervasive*, we must address ethical concerns and biases that may arise:

- The challenge of biased algorithms and the potential for reinforcing societal biases in AI decision-making.
- The importance of transparency and explainability in AI systems, especially in critical applications like healthcare and finance.
- The role of ethics boards and guidelines in ensuring responsible AI development and deployment.

AI's potential for the future is vast, with transformative implications across various domains:

- Advancements in personalized medicine and disease prevention through AI-driven genomics and precision medicine.
- AI's role in revolutionizing education through personalized learning experiences and adaptive tutoring.
- The potential for autonomous vehicles to transform transportation and reduce accidents on our roads.
- Societal implications of AI's role in shaping public policy, ethics, and governance.

It is evident that AI holds enormous potential to drive positive change and advancement across various sectors. However, it also poses significant responsibilities, including addressing ethical considerations and promoting fairness in AI deployment. By working together, we can harness AI's potential to create a better and more equitable future for all. Let us remain proactive in shaping AI's trajectory, ensuring it aligns with human values, and fostering a society that benefits from AI's transformative power.

8. Q&A and Wrap-up

Congratulations! You have completed the introductory journey into the captivating realm of Artificial Intelligence. Throughout this topic, we have laid the foundation for understanding AI's principles, its diverse applications, and the impact it has on our lives. As we conclude this introductory topic, let's recap the key takeaways:

1. What is AI? We have grasped the essence of Artificial Intelligence as the science and engineering of creating intelligent machines capable of simulating human-like intelligence, learning from data, and making decisions.

2. AI Techniques and Approaches: We explored various AI techniques, such as Machine Learning, Natural Language Processing, Computer Vision, Robotics, and Expert Systems. Understanding these techniques provides a glimpse into AI's versatility and real-world applications.

3. Real-World Applications: We have seen how AI is already reshaping industries and daily experiences. From virtual assistants like Siri and Alexa to personalized recommendations on streaming platforms, AI is present in our lives in ways we might not even realize.

4. Ethics in AI: An important aspect we covered is the ethical considerations in AI development and deployment. As AI technologies become more pervasive, addressing bias, transparency, privacy, and accountability is crucial to ensuring AI's responsible and equitable use.

5. The Path Forward: As beginners in AI, this is just the beginning of an exciting journey. The field of AI is continually evolving, and there are endless possibilities to explore, research, and innovate. Continuous learning and staying updated with the latest advancements will enable us to contribute to AI's growth.

6. Collaboration and Interdisciplinary Exploration: AI is a multidisciplinary field that benefits from collaboration across domains. Embracing an interdisciplinary approach allows us to combine AI with other fields, opening new doors for innovation and breakthroughs.

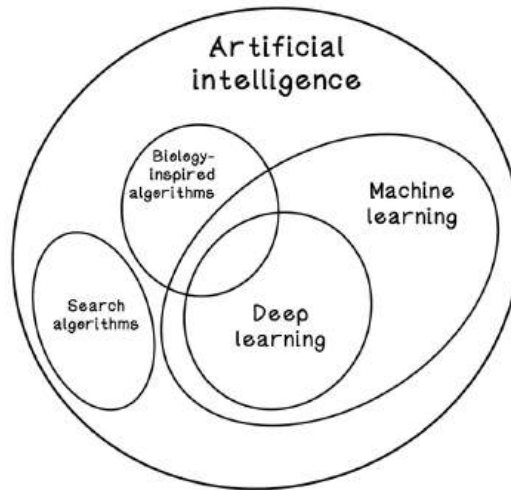
7. AI for Good: As we advance in AI, let us keep in mind the importance of leveraging AI technologies for the betterment of society. AI's potential to address global challenges like healthcare, climate change, and education holds great promise for a brighter future.

In conclusion, your introduction to AI has provided you with the fundamental knowledge and insights to navigate this captivating field. The journey of AI discovery has only just begun, and there is much more to explore. As you delve deeper into the world of Artificial Intelligence, remember to stay curious, be ethical in your pursuits, and embrace the potential of AI as a force for positive change.

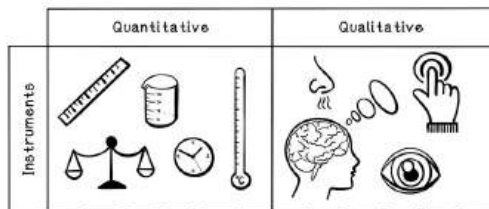
AI is difficult to define. There is no clear consensus.

Look at implementations as being AI-like things that exhibit intelligence.

Many disciplines are encompassed in AI.

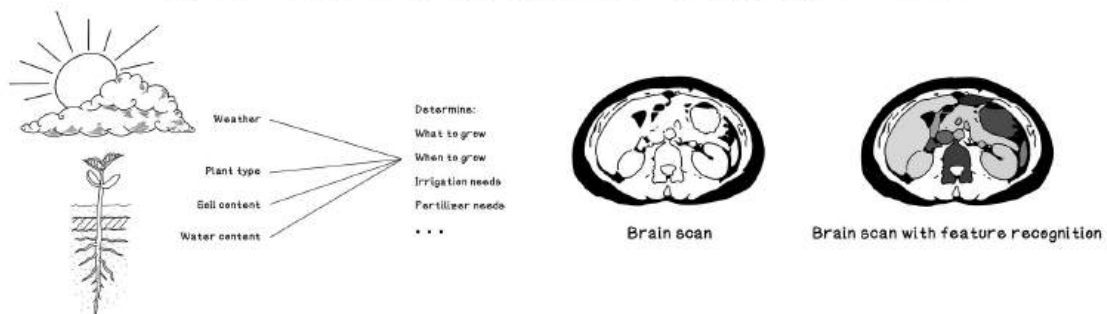


AI implementations almost always have room for error. Be cautious about the consequences of this.



Quality and preparation of data is important.

AI has many uses and applications. Apply your mind!



Be responsible when developing technology.

Thank you for joining us on this introductory adventure into AI. As you continue your AI journey, keep exploring, innovating, and making a positive impact through the power of Artificial Intelligence. The future of AI is in your hands!

Introduction to Artificial Intelligence. Practical Tasks

Task.1 Rock-Paper-Scissors

The Rock-Paper-Scissors game is a simple hand game played between two players. It is often used as a decision-making tool or a playful way to make choices between two individuals. The game involves three hand shapes, each representing one of the three options: "rock," "paper," or "scissors." The rules are straightforward:

1. Rock crushes Scissors: Rock wins against Scissors.
2. Paper covers Rock: Paper wins against Rock.
3. Scissors cut Paper: Scissors win against Paper.

The game follows a cyclic pattern, where each option beats one and loses to another. If both players choose the same option, the game results in a tie or draw.

The Rock-Paper-Scissors game is typically played in a series of rounds. Each player makes their choice by forming one of the hand shapes (rock, paper, or scissors) simultaneously. Then, the choices are revealed, and the winner of the round is determined based on the rules above. The game continues until a predetermined number of rounds are completed, and the player with the most wins is declared the overall winner.

```
!pip install git+https://github.com/mehalyna/cooltest.git
```

```
from cooltest.test_cool_2 import *
```

Pass

Your task is to write the body of function `play_rock_paper_scissors()` in such a way that it would be used to play the game with the computer.

In this version of the function, the player's choice (from the user) and computer's choice (randomly generated) will be taken as input parameter. The winner is determined based on the **Rock-Paper-Scissors** game rules, and the result of the round is displayed. The user can play the game against the computer by providing their choice as input.

```
@test_play_rock_paper_scissors
def play_rock_paper_scissors(player_choice, comp_choice):
    """
    Play a round of the Rock-Paper-Scissors game.

    Parameters:
        player1_choice (str): The choice of Player 1. Should be one
        of 'rock', 'paper', or 'scissors'.
        comp_choice (str): The choice of Computer. Should be one of
        'rock', 'paper', or 'scissors'.

    Returns:
        str: The result of the round. It can be one of the
```

following:

- "Tie" if both players have the same choice.
- "Player 1 wins" if Player 1's choice wins the round.
- "Comp wins" if Computer's choice wins the round.

Example:

```
>>> play_rock_paper_scissors('rock', 'paper')
'Comp wins'
>>> play_rock_paper_scissors('scissors', 'scissors')
'Tie'
"""
valid_choices = {'rock', 'paper', 'scissors'}

# Check if it's a tie

# Determine the winner based on the rules

return result
```

Your next task is to write the body of function `computer_makes_choice` in such a way that it would return the choice of the computer.

The computer's choice will be randomly generated using the `random.choice()` function.

```
import random

@ test_computer_makes_choice
def computer_makes_choice():
    """
    Generates the choice of the computer in the Rock-Paper-Scissors
    game.

    Parameters:
        none.

    Returns:
        str: The choice of the player. Should be one of 'rock',
        'paper', or 'scissors'.

    Example:
        >>> computer_makes_choice()
        'rock'
        >>> computer_makes_choice()
        'scissors'
    """
    valid_choices = {'rock', 'paper', 'scissors'}

    # Generate computer's choice randomly
```



```
return result
```

Write the body of function `play_multiple_rounds(num_rounds)` that will directly call the `play_rock_paper_scissors()` function inside the loop for each round. The game proceeds as before, with the user inputting their choice, and the result of each round is displayed accordingly.

```
def play_multiple_rounds(num_rounds):
    """
    Play multiple rounds of the Rock-Paper-Scissors game against the
    computer.

    Parameters:
        num_rounds (int): The number of rounds to play.

    Returns:
        None

    Example:
        >>> play_multiple_rounds(3)
        Enter your choice (rock/paper/scissors): rock
        Round 1: You chose rock. Computer chose paper. Comp wins

        Enter your choice (rock/paper/scissors): paper
        Round 2: You chose paper. Computer chose paper. Tie

        Enter your choice (rock/paper/scissors): scissors
        Round 3: You chose scissors. Computer chose rock. Comp wins
    """
    valid_choices = {'rock', 'paper', 'scissors'}

    # Implementation of the dialog with the player

    return None

# Example usage:
num_rounds = int(input("Enter the number of rounds to play: "))
play_multiple_rounds(num_rounds)
```

The simplicity and randomness of the Rock-Paper-Scissors game make it a popular choice for resolving disputes or making decisions in a fair and fun way.

Task 2. Hangman game

Create a text-based version of the Hangman game. The computer selects a random word, and the player guesses letters until they solve the word or run out of attempts.

Problem Definition:

- You are tasked with implementing a simple Hangman game in Python.
- The game should select a random word from a predefined list of words.
- The player must guess letters in the word one at a time.

- If the player guesses a correct letter, it is revealed in the word.
- If the player guesses an incorrect letter, a part of the hangman is drawn.
- The player wins if they guess all the letters correctly before the hangman is fully drawn.
- The player loses if the hangman is fully drawn before they guess the word.
- The game should provide feedback to the player after each guess.
- The game continues until the player wins or loses.

Your Hangman game will consist of three functions: `select_random_word`, `display_word`, and `hangman_game`. The `hangman_game` function runs the game loop, while the other functions are used to select a random word and display the word with guessed letters filled in.

Certainly! Here's a description of each of the functions in the Hangman game:

1. `select_random_word` Function:

This function is responsible for selecting a random word from a predefined list of words. It is used at the beginning of the game to choose the word that the player needs to guess.

- Parameters: None
- Returns: A randomly selected word (string).

2. `display_word` Function:

This function takes two arguments: the target word to guess and a list of guessed letters. It generates a string where guessed letters are filled in, and unguessed letters are represented by underscores.

- Parameters:
 - `word(str)`: The target word to guess.
 - `guessed_letters(list)`: List of letters guessed by the player.
- Returns: The word with guessed letters filled in (string).

3. `hangman_game` Function:

This function executes the Hangman game. It handles the game's main logic, including taking user input for letter guesses, checking if the guessed letters are correct, and displaying feedback to the player. The game continues until the player either wins by guessing the word correctly or loses by running out of attempts.

- Parameters: None

Each function serves a specific role in the Hangman game, making the code modular and easier to understand. The `hangman_game` function ties everything together and provides the overall game experience for the player, including welcoming them, managing guesses, and announcing the game's outcome.

```
import random

@test_select_random_word
def select_random_word():
```

```

"""
Selects a random word from a predefined list of words.

Returns:
str: A randomly selected word.
"""
word_list = ["apple", "banana", "cherry", "dog", "elephant",
"flower", "giraffe", "hamburger", "icecream", "jacket"]
return result

@test_display_word
def display_word(word, guessed_letters):
    """
    Displays the word with guessed letters filled in and unguessed
    letters as underscores.

    Args:
    word (str): The target word to guess.
    guessed_letters (list): List of letters guessed by the player.

    Returns:
    str: The word with guessed letters filled in.
    """

    return result

def hangman_game():
    """
    Executes the Hangman game.

    The player must guess letters in a randomly selected word until
    they either win or lose.
    """
    word_to_guess = select_random_word()
    guessed_letters = []
    max_attempts = 6 # Maximum attempts before losing

    print("Welcome to Hangman!")
    print("Try to guess the word. You can make up to 6 wrong
    guesses.")

    return None

# To play the game, call the hangman_game() function:
hangman_game()

```

Topic: AI Ethics, Security, and Confidence

Duration: 1 hour

Agenda:

1. Introduction to AI Ethics, Security, and Confidence
 - a. Understanding the significance of AI ethics
 - b. The importance of security in AI applications
 - c. Building trust and confidence in AI systems
2. Ethical Considerations in AI
 - a. The impact of AI on society and individuals
 - b. Addressing biases in AI algorithms and data
 - c. Ensuring fairness and transparency in AI decision-making
 - d. Privacy concerns and data protection
3. Security in AI Systems
 - a. The role of security in AI model development and deployment
 - b. Protecting AI models from adversarial attacks
 - c. Securing data used for training and inference
 - d. Safeguarding AI systems against unauthorized access
4. Building Confidence in AI Systems
 - a. Explainable AI and interpretability
 - b. Certifications and standards for trustworthy AI
 - c. Communication of AI system limitations to users
5. Real-world Applications and Case Studies
 - a. Examples of AI ethical dilemmas in real-world scenarios
 - b. Instances where security vulnerabilities impacted AI systems
 - c. Building confidence in AI through successful applications
6. Q&A and Wrap-up
 - a. Answering participant questions and clarifications
 - b. Summary of key takeaways from the session

1. Introduction to AI Ethics, Security, and Confidence

Short intro

Welcome to today's session on "AI Ethics, Security, and Confidence." We'll explore the critical aspects of AI development that go beyond just building intelligent systems. We'll delve into understanding the significance of AI ethics, the importance of security in AI applications, and how to build trust and confidence in AI systems. So, let's get started!

AI has become an integral part of our lives, transforming industries and shaping the future. However, with this tremendous potential comes the responsibility to ensure that AI is developed and used ethically. AI ethics involves the examination of the moral implications of AI technology and the principles that guide its development and implementation.

AI ethics encompasses a set of values and guidelines that seek to protect individual rights, promote fairness, transparency, and accountability in AI systems. It addresses questions of bias, discrimination, and the impact of AI on society.

The decisions made by AI systems can have far-reaching consequences, affecting people's lives, privacy, and autonomy. Ethical AI development is essential to prevent unintended harm, ensure fairness, and maintain public trust.

Security is a paramount concern when it comes to AI applications. As AI becomes more powerful and autonomous, safeguarding systems against malicious attacks and ensuring data privacy is crucial.

AI relies heavily on vast datasets, and securing this data is essential to prevent unauthorized access and potential misuse. Data breaches can have severe repercussions, leading to privacy violations and erosion of trust.

Adversarial attacks can exploit vulnerabilities in AI models, leading to incorrect decisions and outcomes. Understanding and mitigating these attacks are vital to maintain the integrity of AI systems.

Trust is the foundation for widespread adoption of AI technologies. Without trust, AI applications may not gain acceptance, hindering their potential benefits to society.

Transparency and Explainability: AI systems must be designed to be transparent and explainable, allowing users to understand the reasoning behind their decisions. Explainable AI helps build trust and facilitates human oversight.

Fairness and Bias Mitigation: Ensuring that AI systems are free from biases and treat all individuals fairly is crucial. Deploying algorithms that actively mitigate biases helps foster trust in AI's ability to make unbiased decisions.

Human-AI Collaboration: Building trust involves striking a balance between human control and AI autonomy. Empowering users to influence AI systems' behavior while leveraging AI's capabilities creates a more trustworthy partnership.

As AI continues to shape our world, it is imperative that we approach its development with ethics, security, and confidence in mind. Understanding the significance of AI ethics allows us

to build systems that positively impact society. Emphasizing security safeguards against potential threats and ensures data privacy. Building trust and confidence in AI systems enables their widespread adoption and acceptance.

2. Ethical Considerations in AI

As AI continues to evolve and influence various aspects of our lives, it's essential to understand its impact on society and individuals. We'll delve into addressing biases in AI algorithms and data, ensuring fairness and transparency in AI decision-making, as well as the important topic of privacy concerns and data protection.

AI has the potential to bring about significant advancements and positive changes. However, its widespread adoption also raises ethical concerns that must be addressed.

AI and Job Displacement: As AI automation becomes more prevalent, it can lead to job displacement and shifts in the job market. We need to consider the impact on the workforce and develop strategies to address potential job transitions.

Social Impact: AI technologies can influence public opinion, spread misinformation, and deepen social divisions. Understanding these implications allows us to develop AI applications that positively contribute to society.

Biases in AI algorithms and data can lead to unfair and discriminatory outcomes. Addressing these biases is essential for building ethical AI systems.

Identifying Biases: AI algorithms may learn biased patterns from historical data, leading to biased decisions. We must identify and understand these biases to prevent perpetuating unfairness.

Mitigating Biases: Implementing techniques such as balanced dataset creation, fairness-aware algorithms, and ongoing evaluation can help mitigate biases and ensure equitable outcomes.

Fairness and transparency are crucial elements for establishing trust in AI systems. Users and society at large must understand how AI arrives at its decisions.

Explainable AI: Developing AI systems that can explain their decision-making processes in human-readable ways fosters trust and enables accountability.

Fair Decision Criteria: Defining and incorporating fairness criteria into AI models ensures that decisions do not discriminate against protected groups.

AI systems often rely on vast amounts of data, raising concerns about privacy and data protection.

Data Collection and Consent: AI developers must be transparent about data collection practices and ensure that users provide informed consent for data usage.

Data Anonymization and Security: Protecting individual privacy involves data anonymization and robust security measures to safeguard sensitive information.

Understanding the impact of AI on society and individuals allows us to shape AI systems that benefit all and avoid harm. By addressing biases, ensuring fairness and transparency, and prioritizing privacy protection, we can build trust and confidence in AI systems.

3. Security in AI Systems

AI model development and deployment, the measures to protect AI models from adversarial attacks, securing data used for training and inference, and safeguarding AI systems against unauthorized access.

Ensuring security in the entire lifecycle of AI systems is vital to prevent potential risks and vulnerabilities.

Secure Coding Practices: Developers must implement secure coding practices when designing AI models, minimizing the chances of introducing vulnerabilities during development.

Regular Security Audits: Conducting regular security audits helps identify and address potential weaknesses and loopholes in AI systems.

Adversarial attacks exploit vulnerabilities in AI models, leading to incorrect decisions and outcomes. Let's explore ways to protect AI models from such attacks.

We must comprehend the various types of adversarial attacks, including evasion and poisoning attacks, to develop effective defense mechanisms.

Designing AI models with robust architectures can improve their resilience against adversarial attacks, making it harder for attackers to manipulate the system.

Data used for training and inference is the backbone of AI systems. Ensuring the security and integrity of this data is crucial.

Data Privacy and Anonymization: Protecting sensitive user information through data anonymization and privacy-preserving techniques prevents unauthorized access to personal data.

Secure Data Storage: Implementing secure data storage practices minimizes the risk of data breaches and unauthorized access to training and inference data.

Unauthorized access to AI systems can lead to data theft, tampering, and misuse. Let's explore ways to safeguard AI systems from unauthorized access.

Authentication and Authorization: Implementing robust authentication and authorization mechanisms ensures that only authorized personnel can access and modify AI systems.

System Monitoring and Intrusion Detection: Continuous monitoring of AI systems and implementing intrusion detection mechanisms help identify and respond to potential security breaches promptly.

As we conclude our session on Security in AI Systems, we recognize the critical role security plays in the development and deployment of AI technologies. By implementing secure coding practices, conducting regular security audits, and understanding and defending against adversarial attacks, we can build robust AI models that inspire confidence and trust.

Additionally, safeguarding data used for training and inference and implementing measures to prevent unauthorized access are vital to protect user privacy and the integrity of AI systems. As AI continues to advance and become more prevalent in our lives, let's remain vigilant in prioritizing security to ensure a safe and trustworthy AI ecosystem.

4. Building Confidence in AI Systems

Explainable AI plays a vital role in building user confidence and trust in AI systems. Let's understand its significance.

Understanding Explainable AI: Explainable AI refers to the ability of AI models to provide clear and understandable explanations for their decisions and actions. It allows users to comprehend why a particular decision was made, leading to increased trust in the system.

Benefits of Interpretability: AI systems that are interpretable enable users to validate the system's decisions, detect biases, and identify potential errors. Interpretability is crucial, particularly in high-stakes applications like healthcare and finance.

Certifications and standards play a significant role in instilling confidence in AI systems by ensuring they meet specific ethical and security requirements.

Trusted AI Frameworks: Various organizations and institutions have developed trusted AI frameworks that outline principles and guidelines for ethical AI development and deployment. Adhering to these frameworks helps build trust with users and stakeholders.

AI Ethics Certifications: AI developers and organizations can obtain certifications that demonstrate their commitment to ethical AI practices. These certifications provide assurance to users that the AI systems they interact with meet ethical standards.

Transparent communication about AI system limitations is essential for managing user expectations and avoiding misunderstandings.

Realistic Expectations: Clearly communicating what AI systems can and cannot do sets realistic expectations for users. It helps users understand the system's limitations and avoids disappointment.

Human-AI Collaboration: Encouraging human-AI collaboration acknowledges that AI systems are not infallible and that human oversight is essential in critical decision-making processes.

In conclusion, building confidence in AI systems is crucial for their widespread acceptance and successful integration into various domains. By incorporating Explainable AI and interpretability, we empower users to understand and trust AI decisions, fostering transparency and accountability.

Certifications and adherence to trustworthy AI frameworks demonstrate a commitment to ethical practices, providing users with the assurance that AI systems meet the highest ethical standards.

Furthermore, effective communication of AI system limitations helps manage user expectations and encourages a harmonious collaboration between humans and AI. By working together to create transparent, accountable, and user-centric AI systems, we can shape a future where AI is a valuable tool for the betterment of society.

5. Real-world Applications and Case Studies

AI is making a tangible impact in various industries and domains. From healthcare to finance, from transportation to education, AI is transforming the way we live and work. Let's dive into some compelling real-world applications and case studies to understand the potential of AI.

5.1. AI in Healthcare

AI is revolutionizing healthcare, providing innovative solutions to improve patient care and medical research.

- **Medical Imaging and Diagnostics:** AI-powered image recognition systems assist radiologists in detecting diseases like cancer from medical images with higher accuracy and efficiency.
- **Personalized Medicine:** AI algorithms analyze patient data to tailor personalized treatment plans, predicting the most effective therapies based on individual characteristics.

5.2. AI in Finance

The financial sector is leveraging AI to enhance decision-making and improve customer experiences.

- **Fraud Detection:** AI-driven fraud detection systems analyze vast amounts of financial data in real-time, helping detect and prevent fraudulent activities.
- **Algorithmic Trading:** AI-powered algorithms execute trades based on market trends and historical data, improving trading efficiency and accuracy.

5.3. AI in Transportation

AI is reshaping transportation, making it safer, more efficient, and sustainable.

- **Autonomous Vehicles:** Self-driving cars equipped with AI sensors and algorithms can navigate roads, reducing accidents and optimizing traffic flow.
- **Traffic Management:** AI-based systems analyze real-time traffic data to optimize traffic signals and reduce congestion in urban areas.

5.4. AI in Education

AI is transforming education by offering personalized learning experiences and streamlining administrative tasks.

- **Personalized Learning:** AI-powered educational platforms adapt content and teaching methods to match individual student needs and learning styles.
- **Administrative Efficiency:** AI automates administrative tasks like grading, scheduling, and student support, freeing up educators' time for more meaningful interactions with students.

As AI continues to advance, it is essential to balance its potential benefits with ethical considerations, transparency, and privacy protection. Collaboration among researchers, policymakers, and industry stakeholders is crucial to harness the full potential of AI while addressing its challenges responsibly.

6. Q&A and Wrap-up

Let's remember that ethical considerations in AI are an ongoing process, requiring collaboration among technologists, policymakers, ethicists, and the public. As AI continues to shape the future, let's commit ourselves to creating AI technologies that align with human values and contribute positively to the betterment of society as a whole. Thank you for your participation today.

AI Ethics. Practical Task

AI Risks Identification Tool

Let's create an interactive and structured *AI Risks Identification Tool* that allows users to categorize the identified risks and provides a summary of the risks under each category. We'll use dictionaries and lists to organize the data.

Below are the descriptions of the functions which will be used in the AI Risks Identification Tool:

`get_risks_and_categories` - This function asks users to input the appropriate category of the potential risks associated with the usage of AI in everyday life. It prompts the user to enter risks and their corresponding categories iteratively until all the elements from `risks_list` will be passed. The risks and categories (entered by the user) are stored in a list of lists, where each category is 0 element of inner list, and the corresponding risks is stored as a 1 element of inner list (eg. `[[category0, risk0], [category1, risk1]], ...`).

- Parameters: This function takes `risks_list` parameter.
- Returns: This function returns `risks_categories` - a list of lists containing the risks and categories.

`identify_risks()` - This function allows users to identify and categorize potential risks associated with the usage of AI in everyday life. The risks and categories are stored in a dictionary, where each category is a key, and the corresponding risks are stored in a list under that key.

- Parameters: This function takes `risks_categories` parameter.
- Returns: This function returns a dictionary containing the categorized risks. The keys of the dictionary are the categories, and the values are lists of risks associated with each category.

`display_risks_summary(risks)` - This function displays a summary of the identified AI risks under each category. It takes the categorized risks dictionary as input and prints the risks in a user-friendly format.

- Parameters: `risks (dict)` - A dictionary containing categorized risks. The keys are the categories, and the values are lists of risks associated with each category.
- Returns: This function does not return any value. It simply prints the summary of identified AI risks.

The `identify_risks()` function organizes the entered risks into a dictionary to keep track of the different risk categories. The `display_risks_summary()` function then takes this dictionary as input and presents a clear summary of the identified AI risks under each category.

These functions enable users to interactively explore and categorize AI risks, making it easier to comprehend the various challenges and concerns associated with the use of AI in everyday life.

You may use these sample data to test your code

Privacy concerns: AI-powered devices may collect personal data, raising concerns about how this data is stored and used.

Category: Privacy

Security vulnerabilities: AI systems could be susceptible to cyberattacks, leading to unauthorized access to personal information.

Category: Security

Bias in AI algorithms: AI algorithms might inadvertently perpetuate biases present in the data they are trained on, leading to unfair or discriminatory outcomes.

Category: Fairness

Bias in AI algorithms may cause biased loan approvals for certain demographic groups.

Category: Fairness

AI system malfunctioning may result in incorrect medical diagnoses.

Category: Safety

AI-driven job automation may lead to unemployment and job displacement.

Category: Social Impact

```
pip install git+https://github.com/mehalyna/cooltest.git
```

```
from cooltest.test_cool_4 import *
```

```
risks_list = [  
    "Privacy concerns: AI-powered devices may collect personal  
    data, raising concerns about how this data is stored and used.",  
    "Security vulnerabilities: AI systems could be susceptible  
    to cyberattacks, leading to unauthorized access to personal  
    information.",  
    "Bias in AI algorithms: AI algorithms might inadvertently  
    perpetuate biases present in the data they are trained on, leading  
    to unfair or discriminatory outcomes.",  
    #you may add more AI Risks to this list  
]
```

```
def get_risks_and_categories(risks_list):
```

```
    """  
    Identify and categorize potential risks of using AI in everyday  
    life.
```

```
    Parameters:  
        risks_list (list): A list of potential risks and categories  
        associated with AI usage.
```

```
    Returns:  
        list: A list containing categorized risks.  
    """
```

```

    print("Welcome to the AI Risks Identification Tool!")
    print("Enter the category of each risk:")

    return result

@test_identify_risks
def identify_risks(risks_categories):
    """
    Identify and categorize potential risks of using AI in everyday
    life.

    Parameters:
        risks_categories (list): A list of potential risks and
        categories associated with AI usage.

    Returns:
        dict: A dictionary containing categorized risks.
    """

    return result

def display_risks_summary(risks):
    """
    Display a summary of identified risks under each category.

    Parameters:
        risks (dict): A dictionary containing categorized risks.
    """
    print("\nAI Risks Summary:\n")

    return None

risks_categories = get_risks_and_categories(risks_list)
risks_data = identify_risks(risks_categories)
display_risks_summary(risks_data)

```

Search Algorithms in AI

Part 1

Duration: 4 hours

Agenda:

1. Introduction to Search Algorithms in AI (Duration: 20 minutes)
 - a. Definition and overview of Search Algorithms in AI
 - b. Importance of search algorithms in AI
 - c. Types of search problems in AI
2. Uninformed Search Algorithms (Duration: 1 hour)
 - a. Depth-First Search (DFS)
 - Explanation of DFS
 - Implementation and examples
 - Advantages and limitations
 - b. Breadth-First Search (BFS)
 - Explanation of BFS
 - Implementation and examples
 - Advantages and limitations
 - c. Comparison between DFS and BFS
 - d. Depth-Limited Search and Iterative Deepening
3. Informed Search Algorithms (Duration: 1 hours)
 - a. Greedy Best-First Search
 - Explanation of Greedy Best-First Search
 - Heuristic functions and their importance
 - Advantages and limitations
 - b. A* Search Algorithm
 - Explanation of A* algorithm
 - Admissible and consistent heuristic functions
 - Optimality and completeness of A*
 - Examples and use cases
 - c. Comparison between Greedy Best-First and A*
4. Local Search Algorithms (Duration: 1 hour)
 - a. Hill-Climbing Algorithm
 - Explanation of Hill-Climbing
 - Types of Hill-Climbing

- Issues with local optima and plateau
- b. Simulated Annealing
 - Explanation of Simulated Annealing
 - Annealing schedule and acceptance probability
 - Overcoming local optima
- 5. Real-world Applications and Case Studies (Duration: 30 minutes)
 - a. AI in gaming and puzzles
 - b. AI in route planning and optimization
 - c. AI in robotics and autonomous systems
- 6. Q&A and Wrap-up (Duration: 10 minutes)
 - a. Questions and concerns
 - b. Summary of key takeaways from the session
 - c. Encouraging further exploration in the field of AI and search algorithms

1. Introduction to Search Algorithms in AI

Today, we will delve into the world of search algorithms, which play a fundamental role in the field of Artificial Intelligence. Throughout this session, we'll cover the definition and overview of search algorithms, discuss their importance in AI, and explore different types of search problems. So, let's get started!

Search Algorithms are a vital part of AI, enabling machines to find solutions to complex problems by systematically exploring various possibilities. In essence, a search algorithm is a step-by-step procedure that helps an AI system navigate through a problem space to find an optimal or satisfactory solution. It's like searching for a needle in a haystack, where the needle represents the solution and the haystack symbolizes all possible combinations or states.

Search algorithms can be understood as a metaphorical "guide" that directs the AI system to explore and evaluate potential paths or solutions, eventually leading it to the best outcome. These algorithms form the backbone of various AI applications, from playing strategic games like chess to finding the shortest route on a map or even solving real-world optimization problems.

Search algorithms are the cornerstone of AI problem-solving. They allow machines to tackle complex tasks that would otherwise be impractical or impossible for humans to handle manually. These algorithms empower AI systems to:

1. *Make informed decisions:* Search algorithms help AI systems make intelligent decisions by evaluating various possibilities and selecting the most promising path.
2. *Optimize resource usage:* In many AI applications, resources like time, memory, or computational power are limited. Search algorithms aid in optimizing the use of these resources, ensuring efficiency and effectiveness in solving problems.

3. *Adapt to changing environments*: AI systems often operate in dynamic and uncertain environments. Search algorithms enable them to adapt and adjust their strategies to cope with these changes.

AI search problems can be categorized based on the problem space and the nature of the search. Some common types of search problems include:

1. *Uninformed Search*: In uninformed search, the AI system explores the problem space blindly, without any knowledge of the problem structure. Breadth-First Search (BFS) and Depth-First Search (DFS) are typical examples of uninformed search algorithms.
2. *Informed Search (Heuristic Search)*: In contrast to uninformed search, informed search algorithms incorporate domain-specific knowledge or heuristics to guide the search process. A* (A-star) algorithm is a well-known example of an informed search algorithm.
3. *Adversarial Search*: Adversarial search is employed in games and scenarios where the AI system competes against one or more opponents. Minimax and Alpha-Beta Pruning are commonly used in adversarial search.

The node structure is usually used in search tasks. The node structure includes a data store and the connection of the current node with other nodes. Nodes form **graphs**, which are actually widely used in artificial intelligence algorithms.

Graphs can be very different. However, in search algorithms, we usually use a specific type of graph - **trees**. Each node in a tree has only one parent node (except for the root node). Trees can also be different: a binary tree is a tree where each node has no more than two children. But now we're going to work with all trees, so we'll leave this point out for now. You can learn more about graphs and trees in the Discrete Mathematics course.

Now let's try to draw our own tree. To do this, we will use the `networkx` library to work with trees and `matplotlib.pyplot` to display the image:

```
import networkx as nx
import matplotlib.pyplot as plt
from networkx.drawing.nx_pydot import graphviz_layout

# Create an empty directed graph
G = nx.DiGraph()

# Add the root node
G.add_node(1)

# Add the first level of nodes
G.add_node(2)
G.add_node(3)

# Add edges to connect the nodes
G.add_edge(1, 2)
G.add_edge(1, 3)

# Add the second level of nodes
G.add_node(4)
```

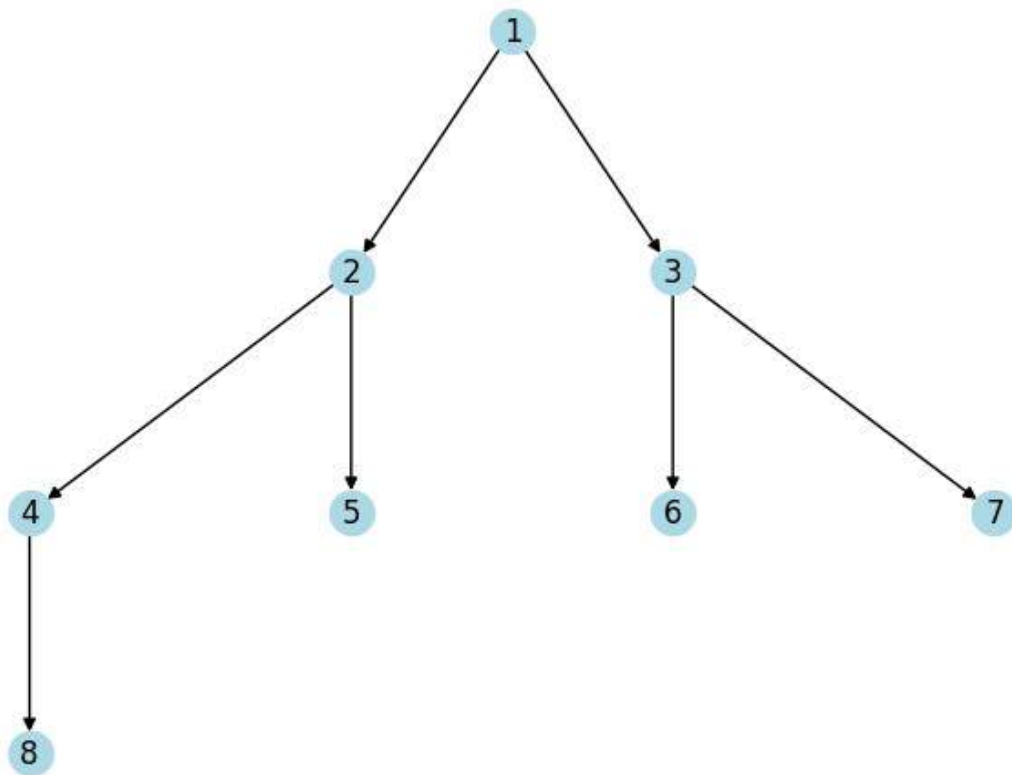


```
G.add_node(5)
G.add_node(6)
G.add_node(7)
G.add_node(8)

# Add edges to connect the nodes
G.add_edge(2, 4)
G.add_edge(2, 5)
G.add_edge(3, 6)
G.add_edge(3, 7)
G.add_edge(4, 8)

# Draw the tree
pos = graphviz_layout(G, prog="dot")
nx.draw(G, pos, with_labels=True, node_color='lightblue')

# Show the tree
plt.show()
```



As you continue your journey into the world of AI, keep in mind that search algorithms serve as a fundamental building block, paving the way for AI systems to excel in problem-solving tasks. Thank you for joining us, and we hope this session has sparked your interest in the fascinating world of search algorithms in AI.

2. Uninformed Search Algorithms (Duration: 1 hour)

In the realm of Artificial Intelligence and problem-solving, search algorithms play a crucial role in guiding machines to find solutions to complex puzzles and challenges. Uninformed search algorithms, also known as blind search algorithms, are a class of techniques where the AI system explores the problem space without possessing any prior knowledge of the problem's structure or characteristics.

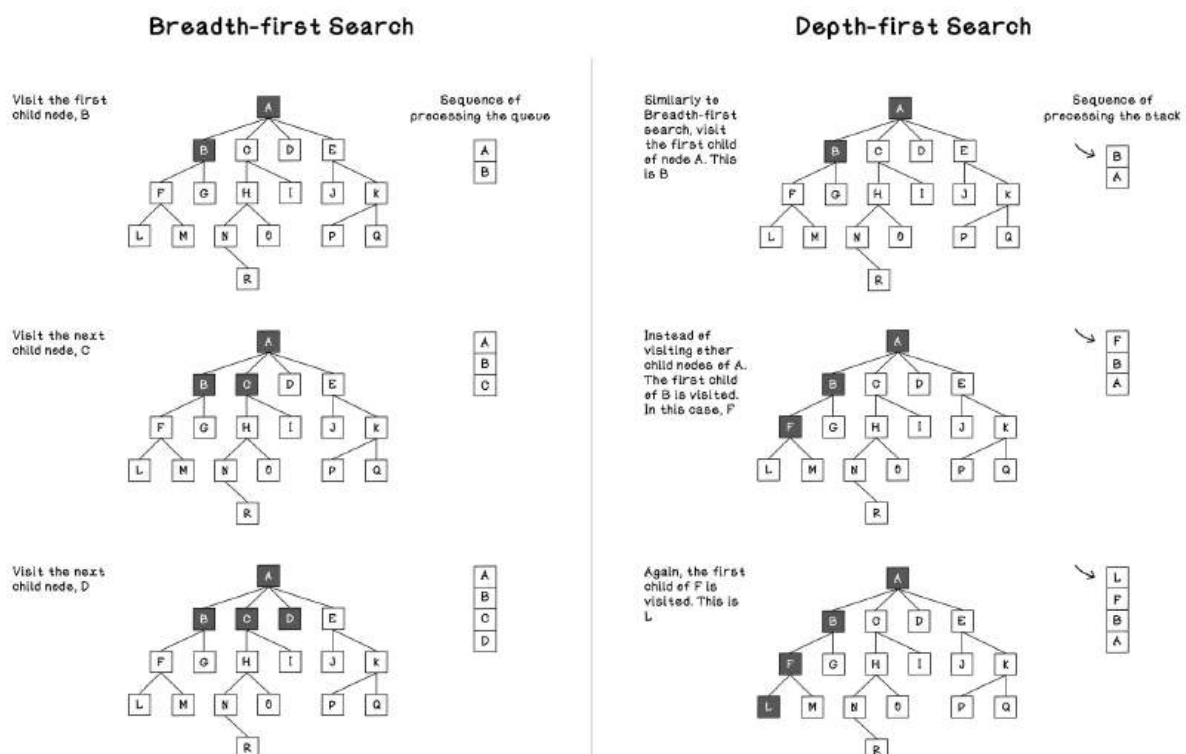
Imagine you are on a treasure hunt in a vast, unknown territory, trying to find the hidden treasure with limited clues. Similarly, uninformed search algorithms enable AI systems to navigate through a problem space without any informed direction, relying solely on systematic exploration to find the desired solution.

In this item, we will explore popular uninformed search strategies, such as Breadth-First Search (BFS) and Depth-First Search (DFS), and understand how these algorithms traverse the problem space in their distinct ways.

Uninformed search algorithms may seem straightforward, but their applications are far-reaching, contributing to various AI problem-solving tasks.

Uninformed search is also known as unguided search, blind search, or brute-force search. Uninformed search algorithms have no additional information about the domain of the problem apart from the representation of the problem which is usually a tree.

Think about when you explore things you want to learn. Some might look at a wide breadth of different topics and learn the basics of each, whereas others might choose one narrow topic and explore its subtopics in depth. This is what Breadth-first Search and Depth-first Search involves, respectively. Depth-first search explores a specific path from the start till it finds a goal at the utmost depth. Breadth-first search explores all options at a specific depth before moving to options deeper in the tree.



2.1. Depth-First Search (DFS)

DFS is an uninformed search algorithm, part of the broader category of graph traversal techniques. It operates by visiting a node or state in the problem space and then recursively explores as far as possible along each branch before backtracking. Think of DFS as akin to exploring a maze by taking a path until you reach a dead-end, and then retracing your steps to explore other paths until you find the exit.

At its core, DFS relies on a stack data structure to maintain the nodes that need to be explored. When a node is visited, its adjacent nodes are pushed onto the stack for further exploration. The algorithm continues this process until all nodes are visited or the goal state is found.

Here's a step-by-step description of the Depth-First Search (DFS) algorithm:

Step 1. Initialize the Stack and Visited Set:

- Create an empty stack to keep track of nodes to explore.
- Create an empty set to store the nodes that have been visited.

Step 2. Push the Start Node onto the Stack and Mark as Visited:

- Start the DFS by pushing the starting node (initial state) onto the stack.
- Mark the starting node as visited by adding it to the visited set.

Step 3. Repeat Until the Stack is Empty:

- While the stack is not empty, do the following steps:

Step 4. Pop a Node from the Stack:

- Remove the top node from the stack. This node will be the current node that we are exploring.

Step 5. Process the Current Node:

- Examine the current node to determine if it is the goal state or if it has any unvisited neighbors that need to be explored.

Step 6. Explore Unvisited Neighbors:

- If the current node has unvisited neighbors, select one of them (based on a specific order, such as left-to-right or based on some heuristic if applicable).
- Push the selected unvisited neighbor onto the stack.
- Mark the neighbor as visited by adding it to the visited set.

Step 7. Backtrack if Necessary:

- If the current node has no unvisited neighbors or is a dead-end (no more possible moves), backtrack by popping the previous node from the stack. This step is crucial in DFS as it allows the algorithm to explore other branches and paths.

Step 8. Repeat the Process:

- Repeat steps 4 to 7 until the stack becomes empty, which means all possible paths have been explored, or the goal state is found.

Step 9. Final Output:

- If the goal state is found, the algorithm terminates, and the path or solution is obtained.
- If the stack becomes empty without finding the goal state, then no solution exists for the given problem.

It's important to note that the DFS algorithm explores a single branch in depth before backtracking, making it a suitable choice for solving problems with large state spaces and situations where finding any solution quickly is more important than finding the optimal solution.

Advantages of Depth-First Search:

1. **Memory Efficiency:** DFS has relatively low memory requirements as it explores a single branch in-depth before backtracking. It is well-suited for problems with large state spaces, where memory consumption is a concern.
2. **Completeness (In Finite Spaces):** In problems with finite state spaces, DFS is guaranteed to find a solution if one exists. This property holds because it explores all possible paths.
3. **Applicability in Certain Scenarios:** DFS is particularly useful in scenarios where finding any solution quickly is more important than finding the optimal solution. It is commonly applied in pathfinding and maze-solving scenarios.

Limitations of Depth-First Search:

1. **Completeness (In Infinite Spaces):** DFS may not be complete in infinite state spaces or graphs with loops. In such cases, it can get trapped in cycles and fail to find a solution even if one exists.
2. **Lack of Optimal Solution:** DFS does not guarantee finding the optimal solution. It might find a solution faster, but that solution may not be the most optimal one.
3. **Memory Usage (In Infinite Spaces):** In infinite state spaces or deeply branching problems, DFS may consume a lot of memory due to its depth-first exploration, potentially leading to stack overflow.

DFS can be used in various scenarios, including graph traversal, pathfinding, puzzle-solving, and more. However, it has its limitations, such as potentially getting stuck in infinite loops (infinite state spaces) or not guaranteeing an optimal solution.

By carefully implementing the steps mentioned above, the Depth-First Search algorithm provides a systematic approach to explore problem spaces effectively and find solutions in various AI applications.

Python Implementation of Depth-First Search:

```
def dfs(graph, node, visited):
    if node not in visited:
        print("Visiting node:", node)
        visited.add(node)
        for neighbor in graph[node]:
            dfs(graph, neighbor, visited)
```

Example Usage:

```
graph = {
    'A': ['B', 'C', 'E'],
    'B': ['D', 'F'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

visited_nodes = set()
dfs(graph, 'A', visited_nodes)

Visiting node: A
Visiting node: B
Visiting node: D
Visiting node: F
Visiting node: C
Visiting node: E
```

2.2. Breadth-First Search (BFS)

In the realm of Artificial Intelligence and algorithms, BFS is a powerful technique for traversing and exploring problem spaces in a systematic manner. Unlike Depth-First Search, BFS focuses on exploring all neighboring nodes at the current level before moving to deeper levels, resembling the ripples formed when a stone is dropped in water.

Breadth-First Search is an uninformed search algorithm that operates by systematically exploring the immediate neighbors of a given node before moving on to their respective neighbors. Think of BFS as exploring all the nearby possibilities before venturing into deeper levels. This approach ensures that BFS searches for the shortest path or the shallowest solution first.

BFS uses a queue data structure to maintain the nodes to be explored. The algorithm starts from the initial state (start node) and expands the search to all neighboring nodes at the same level, then moves to their neighbors in the subsequent level until the goal state is found or all possible states are explored.

Step-by-Step Algorithm for Breadth-First Search (BFS):

Step 1. Initialize the Queue and Visited Set:

- Create an empty queue to store nodes to explore.
- Create an empty set to keep track of visited nodes.

Step 2. Enqueue the Start Node and Mark as Visited:

- Begin the BFS by enqueueing the starting node (initial state) into the queue.
- Mark the starting node as visited by adding it to the visited set.

Step 3. Repeat Until the Queue is Empty:

- While the queue is not empty, do the following steps:

Step 4. Dequeue a Node from the Queue:

- Remove the front node from the queue. This node will be the current node that we are exploring.

Step 5. Process the Current Node:

- Examine the current node to determine if it is the goal state or if it has any unvisited neighbors that need to be explored.

Step 6. Explore Unvisited Neighbors:

- If the current node has unvisited neighbors, enqueue them into the queue.
- Mark the neighbors as visited by adding them to the visited set.

Step 7. Repeat the Process:

- Repeat steps 4 to 6 until the queue becomes empty, which means all nodes at the current level and their neighbors have been explored.

Step 8. Final Output:

- If the goal state is found, the algorithm terminates, and the path or solution is obtained.
- If the queue becomes empty without finding the goal state, then no solution exists for the given problem.

Let's explore a simple Python implementation of BFS for a basic graph traversal scenario. Suppose we have a graph represented as an adjacency list:

```
from collections import deque

def bfs(graph, start):
    queue = deque([start])
    visited = set([start])

    while queue:
        current_node = queue.popleft()
        print("Visiting node:", current_node)

        for neighbor in graph[current_node]:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)

# Example Usage:
graph = {
    'A': ['B', 'C', 'E'],
    'B': ['D', 'F'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

bfs(graph, 'A')
```

```
Visiting node: A
Visiting node: B
Visiting node: C
Visiting node: E
Visiting node: D
Visiting node: F
```

Advantages of Breadth-First Search:

1. **Guaranteed Optimal Solution:** BFS guarantees finding the optimal solution (shortest path) for problems with unit path costs, as it explores nodes level by level.
2. **Completeness (In Finite Spaces):** For problems with finite state spaces, BFS is complete, meaning it will find a solution if one exists.
3. **Suitable for Pathfinding:** BFS is particularly useful for finding the shortest path in unweighted graphs, making it ideal for pathfinding scenarios.

Limitations of Breadth-First Search:

1. **Memory Usage:** BFS may consume significant memory, especially in graphs with deep levels or an infinite state space.
2. **Inefficiency (In Infinite Spaces):** In infinite state spaces, BFS may take a long time to explore all nodes at a given level before moving to the next level.

2.3. Comparison between Depth-First Search (DFS) and Breadth-First Search (BFS)

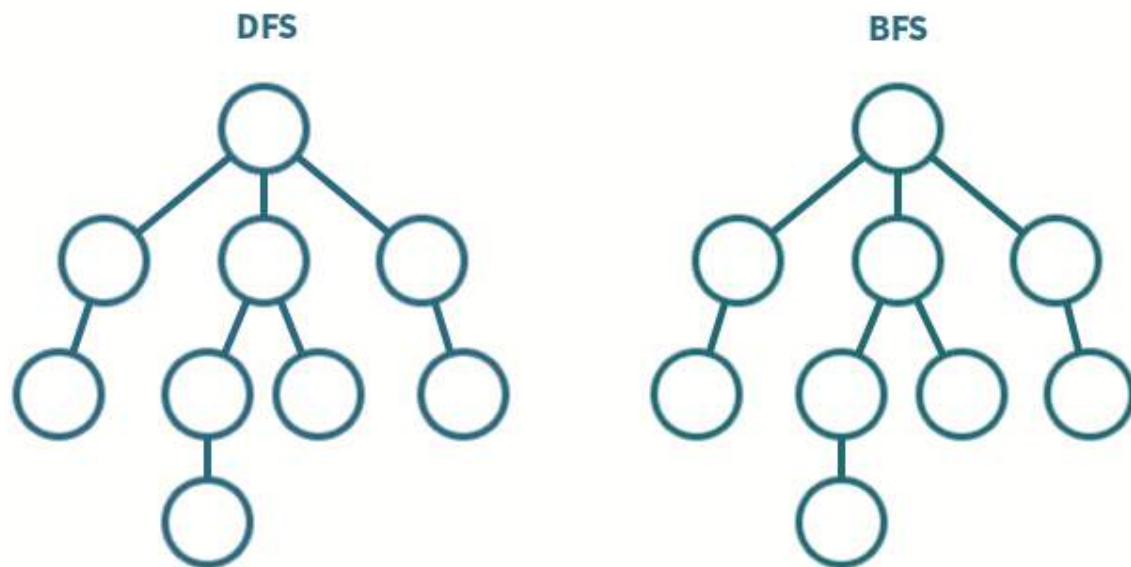
Depth-First Search (DFS) and Breadth-First Search (BFS) are two fundamental search algorithms used in Artificial Intelligence and other fields for problem-solving and graph traversal. While both algorithms aim to explore and find solutions to problems, they have distinct characteristics that make them suitable for different scenarios. Let's compare DFS and BFS based on several key aspects:

Aspect / Algorithm	DFS	BFS
Approach	Explores as far as possible along each branch before backtracking. It delves deeply into the problem space, focusing on exploring one branch at a time, going as deep as possible before exploring other branches.	Explores all neighbors of the current node before moving on to their respective neighbors. It systematically explores nodes level by level, moving outward from the start node, and forming concentric circles like ripples in water.
Order of Exploration	Follows the Last-In-First-Out (LIFO) order of exploring nodes using a stack data structure	Follows the First-In-First-Out (FIFO) order of exploring nodes using a queue data structure
Completeness	In finite state spaces, DFS is complete, meaning it will	BFS is complete in finite state spaces and will always

Aspect / Algorithm	DFS	BFS
	find a solution if one exists. However, in infinite state spaces or graphs with loops, it may not terminate or find a solution if one exists.	find the shortest path in unweighted graphs if one exists.
Memory Usage	DFS typically requires less memory compared to BFS because it explores one branch at a time and uses a stack for backtracking. However, it may consume significant memory in graphs with deep levels or infinite state spaces.	BFS generally requires more memory compared to DFS as it needs to store all nodes at the current level in the queue before moving to the next level.
Optimal Solution	DFS does not guarantee finding the optimal solution, especially in graphs with varying edge weights or costs. It might find a solution faster, but that solution may not be the shortest or most optimal one.	BFS guarantees finding the optimal solution (shortest path) for problems with unit path costs, as it explores nodes level by level.
Use Cases	DFS is well-suited for problems where finding any solution quickly is more important than finding the optimal one. It is commonly used in maze-solving, puzzle-solving, and depth-limited search scenarios.	BFS is ideal for finding the shortest path in unweighted graphs and is commonly used in pathfinding, puzzle-solving, and network traversal scenarios.

In summary, DFS and BFS offer different strategies for exploring problem spaces and finding solutions. DFS is more memory-efficient and suitable for problems where finding any solution quickly is crucial, while BFS guarantees finding the optimal solution but may consume more memory. The choice between DFS and BFS depends on the specific problem requirements, the nature of the problem space, and the goals of the search process.

Visualization of DFS and BFS Algorithms. Resource <https://medium.com/analytics-vidhya/a-quick-explanation-of-dfs-bfs-depth-first-search-breadth-first-search-b9ef4caf952c>



2.4. Depth-Limited Search and Iterative Deepening

Depth-Limited Search is a variation of Depth-First Search (DFS) that imposes a maximum depth or level on the search, preventing it from exploring beyond a specified depth limit. It combines the advantages of DFS's memory efficiency with the completeness of BFS within a limited depth range. DLS is particularly useful in scenarios with limited memory resources or infinite state spaces, where traditional DFS may get stuck in infinite loops.

Step-by-Step Algorithm for Depth-Limited Search (DLS):

Step 1. Initialize the Stack and Visited Set:

- Create an empty stack to store nodes to explore.
- Create an empty set to keep track of visited nodes.

Step 2. Push the Start Node onto the Stack and Mark as Visited:

- Start the DLS by pushing the starting node (initial state) onto the stack.
- Mark the starting node as visited by adding it to the visited set.

Step 3. Set the Depth Limit:

- Define a depth limit, denoted as "depth_limit," which specifies the maximum depth the search is allowed to explore.

Step 4. Repeat Until the Stack is Empty or Depth Limit is Reached:

- While the stack is not empty and the depth limit has not been reached, do the following steps:

Step 5. Pop a Node from the Stack:

- Remove the top node from the stack. This node will be the current node that we are exploring.

Step 6. Process the Current Node:

- Examine the current node to determine if it is the goal state or if it has any unvisited neighbors within the depth limit that need to be explored.

Step 7. Explore Unvisited Neighbors within Depth Limit:

- If the current node has unvisited neighbors at a depth less than or equal to the depth limit, select one of them.
- Push the selected unvisited neighbor onto the stack.
- Mark the neighbor as visited by adding it to the visited set.

Step 8. Repeat the Process:

- Repeat steps 5 to 7 until the stack becomes empty or the depth limit is reached.

Step 9. Final Output:

- If the goal state is found, the algorithm terminates, and the path or solution is obtained.
- If the depth limit is reached without finding the goal state, increment the depth limit and repeat steps 2 to 8 until the goal is found or deemed unreachable within the expanded depth limits.

Below is an example of Depth-Limited Search (DLS) implemented in Python. In this example, we'll perform DLS on a simple binary tree to find a target value within a specified depth limit.

```
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

def depth_limited_search(node, target, depth_limit):
    if node is None:
        return False

    if node.val == target:
        return True

    if depth_limit == 0:
        return False

    # Perform DLS on left and right subtrees with reduced depth
    # limit
    return depth_limited_search(node.left, target, depth_limit - 1)
    or depth_limited_search(node.right, target, depth_limit - 1)

# Example Binary Tree:
#           1
#        /  \
#       2    3
#      / \   \
#     4  5   6
```

```

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.right = TreeNode(6)

# DLS Example
target_value = 6
depth_limit = 2

result = depth_limited_search(root, target_value, depth_limit)

if result:
    print(f"Target value {target_value} found within depth limit {depth_limit}.")
else:
    print(f"Target value {target_value} not found within depth limit {depth_limit}.")

Target value 6 found within depth limit 2.

```

In this example, we define a `TreeNode` class to represent each node of the binary tree. The `depth_limited_search` function takes the current node, the target value to be found, and the depth limit as input parameters.

The function recursively explores the binary tree, checking if the current node's value matches the target value. If not, it performs DLS on the left and right subtrees with a reduced depth limit until the depth limit reaches 0 or the target value is found.

For the given binary tree with the depth limit of 2, the function searches for the target value 5. The output will indicate whether the target value was found within the specified depth limit or not.

Keep in mind that this example is a simplified demonstration of DLS on a binary tree. In practice, DLS can be applied to more complex problem spaces and search scenarios.

Iterative Deepening is an intelligent search strategy that uses Depth-Limited Search (DLS) with an incrementally increasing depth limit. It repeatedly applies DLS with increasing depth limits until the goal state is found or a specified maximum depth is reached. This approach combines the benefits of BFS's completeness and DFS's memory efficiency.

Step-by-Step Algorithm for Iterative Deepening:

Step 1. Set Initial Depth Limit:

- Define an initial depth limit, denoted as "initial_depth_limit."

Step 2. Repeat Until Goal is Found or Max Depth is Reached:

- While the goal state is not found and the current depth limit is less than or equal to the maximum depth limit, do the following steps:

Step 3. Perform Depth-Limited Search:

- Apply Depth-Limited Search (DLS) with the current depth limit on the problem space.

Step 4. If Goal is Found, Terminate:

- If the goal state is found during DLS, terminate the search and output the path or solution.

Step 5. Increment Depth Limit:

- If the goal state is not found within the current depth limit, increment the depth limit.

Step 6. Repeat the Process:

- Repeat steps 2 to 5 until the goal state is found or the maximum depth limit is reached.

Iterative Deepening effectively combines the completeness of BFS with the memory efficiency of DFS. It allows the search to explore deeper levels progressively, avoiding the need to store a massive search tree in memory, making it particularly useful in scenarios with limited resources and infinite state spaces.

Below is an example of Iterative Deepening implemented in Python. In this example, we'll use Iterative Deepening to find a target value in a binary tree. The depth limit will be incrementally increased until the target value is found or a specified maximum depth is reached.

```
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

def depth_limited_search(node, target, depth_limit):
    if node is None:
        return False

    if node.val == target:
        return True

    if depth_limit == 0:
        return False

    # Perform DLS on left and right subtrees with reduced depth
    # limit
    return depth_limited_search(node.left, target, depth_limit - 1)
    or depth_limited_search(node.right, target, depth_limit - 1)

def iterative_deepening(root, target, max_depth):
    for depth_limit in range(max_depth + 1):
        if depth_limited_search(root, target, depth_limit):
            return True
    return False

# Example Binary Tree:
#       1
#      / \
#     2   3
#    / \   \
#   /  \   \
```

```

#      4    5    6

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.right = TreeNode(6)

# Iterative Deepening Example
target_value = 5
max_depth = 3

result = iterative_deepening(root, target_value, max_depth)

if result:
    print(f"Target value {target_value} found within maximum depth
of {max_depth}.")
else:
    print(f"Target value {target_value} not found within maximum
depth of {max_depth}.")

```

In this example, we define a `TreeNode` class to represent each node of the binary tree. The `depth_limited_search` function is the same as the one used in the Depth-Limited Search example. The `iterative_deepening` function takes the root of the binary tree, the target value to be found, and the maximum depth as input parameters.

The function iteratively performs Depth-Limited Search (DLS) with increasing depth limits, starting from 0 up to the specified maximum depth. If the target value is found within any of the depth limits, the function returns `True`. Otherwise, if the maximum depth is reached without finding the target value, the function returns `False`.

For the given binary tree with a maximum depth of 3, the function searches for the target value 5. The output will indicate whether the target value was found within the specified maximum depth or not.

Iterative Deepening is a useful technique to combine the advantages of both Breadth-First Search (BFS) and Depth-First Search (DFS) by exploring deeper levels progressively while avoiding the need to store a massive search tree in memory.

2.5. Realisation of BFS and DFS in Graphs

Let's make some preparations: create a graph and import the necessary libraries. We'll also announce what we're going to search for and where we're going to find it:

```

import networkx as nx
import matplotlib.pyplot as plt
from networkx.drawing.nx_pydot import graphviz_layout

# Define the graph
G = nx.Graph()
G.add_nodes_from([1, 2, 3, 4, 5, 6, 7])
G.add_edge(1, 2)
G.add_edge(1, 3)

```

```
G.add_edge(2, 4)
G.add_edge(2, 5)
G.add_edge(3, 6)
G.add_edge(3, 7)

# Define the start and goal nodes
start_node = 1
goal_node = 2
```

Now let's implement the search in depth: to do this, declare the array of vertices we have already passed (to avoid repetition) and the stack on which we will move (exactly the stack, because we are moving in depth):

```
# Define the depth-first search function
def dfs(node):
    print('DFS:')
    visited = []
    stack = [node]
    while stack:
        current_node = stack.pop()
        print('Node', current_node)
        if current_node == goal_node:
            visited.append(current_node)
            return list(visited)
        visited.append(current_node)
        for neighbor in G.neighbors(current_node):
            if neighbor not in visited:
                stack.append(neighbor)
    return None
```

Unlike the previous algorithm, the breadth search does not have a stack on which to move, but a queue to which we add vertices that will be considered later:

```
# Define the breadth-first search function
def bfs(node):
    print('BFS:')
    visited = []
    queue = [[node]]
    while queue:
        path = queue.pop(0)
        current_node = path[-1]
        print('Node', current_node)
        if current_node == goal_node:
            visited.append(current_node)
            return list(path)
        visited.append(current_node)
        for neighbor in G.neighbors(current_node):
            if neighbor not in visited:
                new_path = path.copy()
                new_path.append(neighbor)
                queue.append(new_path)
    return None
```

Perform both of them:

```
# Perform the depth-first search
dfs_path = dfs(start_node)
# Perform the breadth-first search
bfs_path = bfs(start_node)

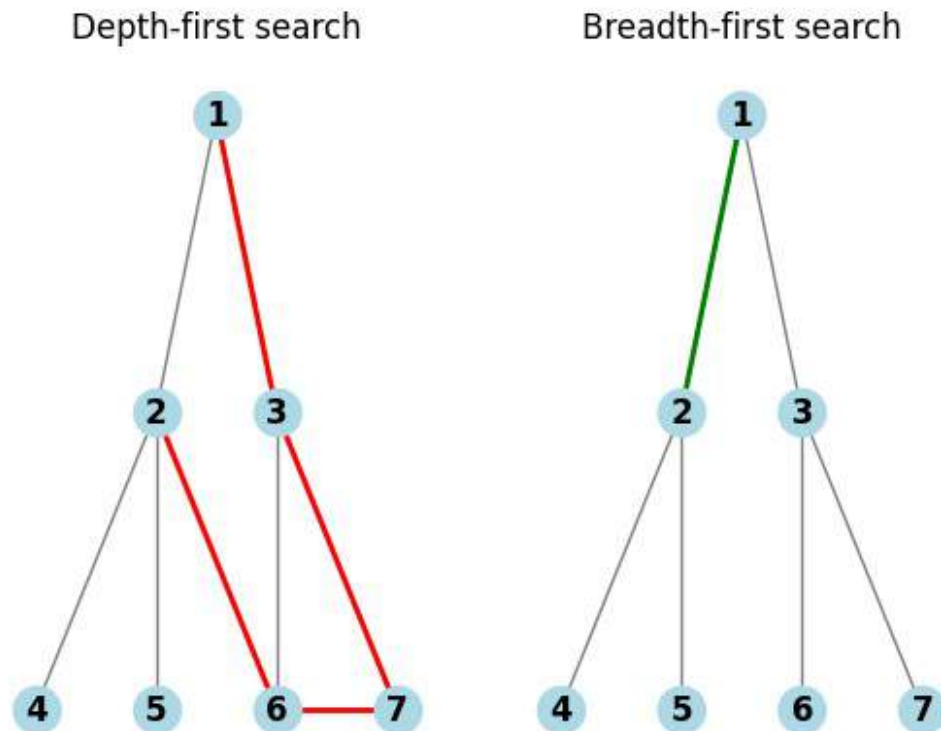
DFS:
Node 1
Node 3
Node 7
Node 6
Node 2
BFS:
Node 1
Node 2
```

And also visualize them using `matplotlib.pyplot`:

```
# Draw the graph with the searched path highlighted for the depth-
first search
plt.subplot(1, 2, 1)
pos = graphviz_layout(G, prog="dot")
nx.draw(G, pos, with_labels=True, font_weight='bold',
node_color='lightblue', edge_color='gray')
if dfs_path:
    dfs_edges = list(zip(dfs_path[:-1], dfs_path[1:]))
    nx.draw_networkx_edges(G, pos, edgelist=dfs_edges,
edge_color='red', width=2)
plt.title('Depth-first search')

# Draw the graph with the searched path highlighted for the breadth-
first search
plt.subplot(1, 2, 2)
pos = graphviz_layout(G, prog="dot")
nx.draw(G, pos, with_labels=True, font_weight='bold',
node_color='lightblue', edge_color='gray')
if bfs_path:
    bfs_edges = list(zip(bfs_path[:-1], bfs_path[1:]))
    nx.draw_networkx_edges(G, pos, edgelist=bfs_edges,
edge_color='green', width=2)
plt.title('Breadth-first search')

# Show the two plots
plt.show()
```



As you can see, to find a node with a value of 2, we went through half of the tree when searching in depth. And when searching in width, we found it literally as the second value (we could have found it as the third if we started with 3). Accordingly, we could have also guessed with the depth search and gotten to it right away. But when searching in trees with many nodes, we can't rely on luck.

Now let's change the search target and see what happens:

```
goal_node = 7

dfs_path = dfs(start_node)
bfs_path = bfs(start_node)
# Draw the graph with the searched path highlighted for the depth-
# first search
plt.subplot(1, 2, 1)
pos = graphviz_layout(G, prog="dot")
nx.draw(G, pos, with_labels=True, font_weight='bold',
node_color='lightblue', edge_color='gray')
if dfs_path:
    dfs_edges = list(zip(dfs_path[:-1], dfs_path[1:]))
    nx.draw_networkx_edges(G, pos, edgelist=dfs_edges,
edge_color='red', width=2)
plt.title('Depth-first search')

# Draw the graph with the searched path highlighted for the breadth-
# first search
plt.subplot(1, 2, 2)
pos = graphviz_layout(G, prog="dot")
nx.draw(G, pos, with_labels=True, font_weight='bold',
node_color='lightblue', edge_color='gray')
```



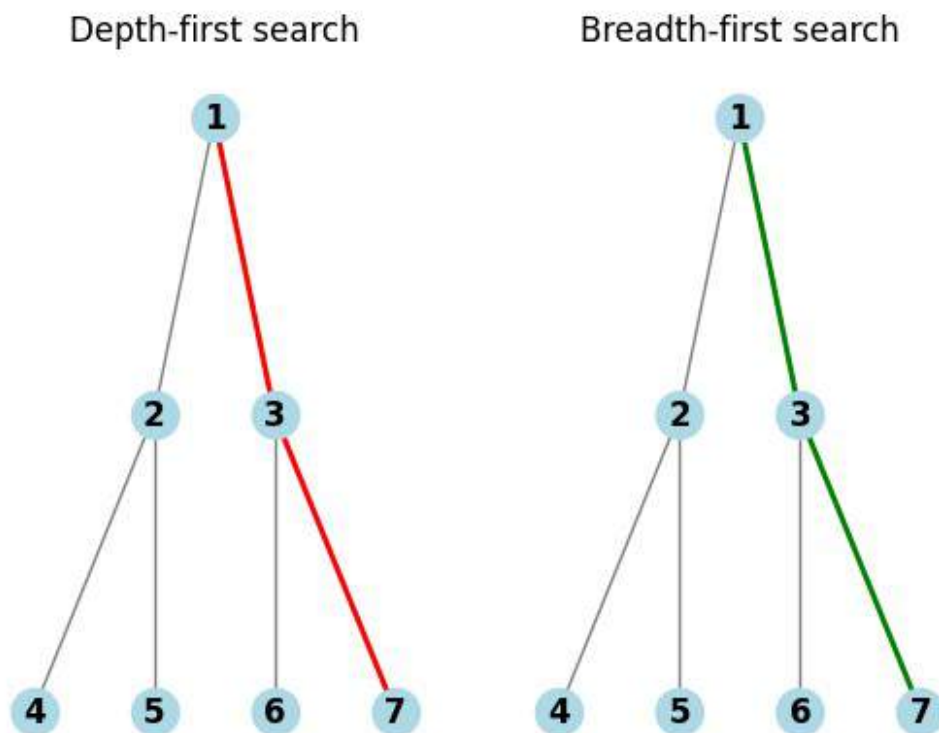
```

if bfs_path:
    bfs_edges = list(zip(bfs_path[:-1], bfs_path[1:]))
    nx.draw_networkx_edges(G, pos, edgelist=bfs_edges,
edge_color='green', width=2)
plt.title('Breadth-first search')

# Show the two plots
plt.show()

DFS:
Node 1
Node 3
Node 7
BFS:
Node 1
Node 2
Node 3
Node 4
Node 5
Node 6
Node 7

```



2.6. Tower of Hanoi

It seems like a child's game, but in fact it is one of the classic tasks in programming. One of the ways to solve it can be a well-known search, in particular in width and depth.

The Tower of Hanoi is a classic mathematical puzzle or game that consists of three pegs and a number of disks of different sizes. The objective of the puzzle is to move all the disks from

one peg (the source) to another peg (the destination), using the third peg (the auxiliary), while following a set of rules:

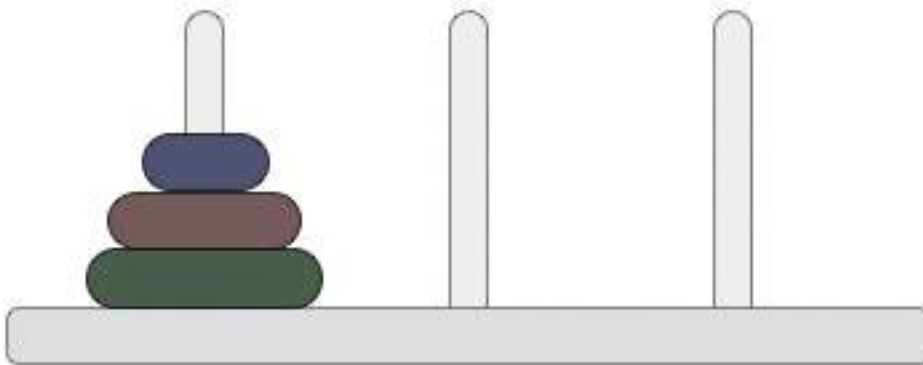
Only one disk can be moved at a time. Each move consists of taking the top disk from one of the pegs and placing it on top of another peg or on an empty peg. A larger disk cannot be placed on top of a smaller disk.

The puzzle starts with all the disks stacked on one peg in decreasing order of size, with the largest disk at the bottom and the smallest disk at the top. The goal is to move all the disks to another peg, maintaining the same order of sizes.

The minimum number of moves required to solve the Tower of Hanoi puzzle with n disks is $2^n - 1$.

Resource <https://ioecapsule.com/tower-of-hanoi-algorithm-and-implementation/>

Step: 0



```
def hanoi_dfs(n):
    start_state = tuple([tuple(range(n, 0, -1)), (), ()])
    goal_state = (), (), tuple(range(n, 0, -1))

    stack = [(start_state, [])]
    visited = set([start_state])

    while stack:
        state, path = stack.pop()

        if state == goal_state:
            return path

        for move in legal_moves(state):
            new_towers = [list(tower) for tower in state]
            new_towers[move[1]].append(new_towers[move[0]].pop())

            new_state = tuple([tuple(tower) for tower in
new_towers])
            if new_state not in visited:
                visited.add(new_state)
                stack.append((new_state, path + [move]))
```

```

    return None

def hanoi_bfs(n):
    start_state = tuple([tuple(range(n, 0, -1)), (), ()])
    goal_state = ((), (), tuple(range(n, 0, -1)))

    queue = [(start_state, [])]
    visited = set([start_state])

    while queue:
        state, path = queue.pop(0)

        if state == goal_state:
            return path

        for move in legal_moves(state):
            new_towers = [list(tower) for tower in state]
            new_towers[move[1]].append(new_towers[move[0]].pop())

            new_state = tuple([tuple(tower) for tower in
new_towers])
            if new_state not in visited:
                visited.add(new_state)
                queue.append((new_state, path + [move]))

    return None

def legal_moves(state):
    moves = []
    for i in range(len(state)):
        for j in range(len(state)):
            if i != j and (state[i] and (not state[j] or state[i][-
1] < state[j][-1])):
                moves.append((i, j))
    return moves

def print_solution(moves):
    if moves is None:
        print("No solution found.")
    else:
        for move in moves:
            print("Move disk from tower", move[0] + 1, "to tower",
move[1] + 1)

n = 2 # Number of disks
print("DFS:")
dfs_moves = hanoi_dfs(n)
print_solution(dfs_moves)

print("\nBFS:")
bfs_moves = hanoi_bfs(n)
print_solution(bfs_moves)

```

DFS:

```
Move disk from tower 1 to tower 3  
Move disk from tower 1 to tower 2  
Move disk from tower 3 to tower 1  
Move disk from tower 2 to tower 3  
Move disk from tower 1 to tower 3
```

BFS:

```
Move disk from tower 1 to tower 2  
Move disk from tower 1 to tower 3  
Move disk from tower 2 to tower 3
```

In conclusion, uninformed search algorithms provide fundamental strategies for exploring problem spaces in Artificial Intelligence. These algorithms, such as Depth-First Search (DFS) and Breadth-First Search (BFS), operate without prior knowledge of the problem's structure, systematically evaluating potential solutions to find the desired outcome. While uninformed search lacks the optimality and efficiency guarantees of some other approaches, its simplicity and versatility make it a valuable tool in various AI applications. By applying uninformed search algorithms intelligently and understanding their strengths and limitations, we can efficiently navigate complex problem spaces and arrive at satisfactory solutions.

Search Algorithm in AI

Part 2

Duration: 4 hours

Agenda:

1. Introduction Search Algorithms in AI (Duration: 20 minutes)
 - a. Definition and overview of AI
 - b. Importance of search algorithms in AI
 - c. Types of search problems in AI
2. Uninformed Search Algorithms (Duration: 1 hour)
 - a. Depth-First Search (DFS)
 - Explanation of DFS
 - Implementation and examples
 - Advantages and limitations
 - b. Breadth-First Search (BFS)
 - Explanation of BFS
 - Implementation and examples
 - Advantages and limitations
 - c. Comparison between DFS and BFS
 - d. Depth-Limited Search and Iterative Deepening
3. Informed Search Algorithms (Duration: 1 hours)
 - a. Greedy Best-First Search
 - Explanation of Greedy Best-First Search
 - Heuristic functions and their importance
 - Advantages and limitations
 - b. A* Search Algorithm
 - Explanation of A* algorithm
 - Admissible and consistent heuristic functions
 - Optimality and completeness of A*
 - Examples and use cases
 - c. Comparison between Greedy Best-First and A*
4. Local Search Algorithms (Duration: 1 hour)
 - a. Hill-Climbing Algorithm
 - Explanation of Hill-Climbing
 - Types of Hill-Climbing

- Issues with local optima and plateau
- b. Simulated Annealing
 - Explanation of Simulated Annealing
 - Annealing schedule and acceptance probability
 - Overcoming local optima
- 5. Real-world Applications and Case Studies (Duration: 30 minutes)
 - a. AI in gaming and puzzles
 - b. AI in route planning and optimization
 - c. AI in robotics and autonomous systems
- 6. Q&A and Wrap-up (Duration: 10 minutes)
 - a. Questions and concerns
 - b. Summary of key takeaways from the session
 - c. Encouraging further exploration in the field of AI and search algorithms

3. Informed Search Algorithms (Duration: 1 hours)

Informed search algorithms, also known as heuristic search algorithms, elevate the problem-solving capabilities of Artificial Intelligence by incorporating domain-specific knowledge or heuristics. Unlike uninformed search algorithms that operate blindly, informed search algorithms intelligently guide the search process, making more informed decisions to explore promising paths first. These algorithms employ heuristic functions that estimate the potential cost or distance to the goal state, providing valuable insights for evaluating the desirability of different actions or states. By leveraging this informed guidance, AI systems can navigate complex problem spaces more efficiently, resulting in faster convergence towards optimal solutions. Informed search algorithms play a pivotal role in various AI applications, including pathfinding, game playing, route planning, and optimization problems, enhancing their effectiveness and applicability in real-world scenarios.

3.1. Greedy Best-First Search - Navigating Efficiently with Heuristic Guidance

In the realm of informed search algorithms, Greedy Best-First Search stands out as a powerful technique that combines heuristic guidance with efficiency in exploration. Unlike uninformed search algorithms, Greedy Best-First Search intelligently selects the most promising path based on heuristic evaluation, making it well-suited for optimization problems and situations where finding a quick but not necessarily optimal solution is desirable. Today, we'll delve into the step-by-step algorithm of Greedy Best-First Search, explore heuristic functions and their significance, provide a Python code example, and discuss its advantages and limitations.

Greedy Best-First Search explores the problem space by prioritizing the nodes that appear most promising according to a heuristic evaluation function. This function, often denoted as $h(n)$, estimates the potential cost or distance from a node to the goal state. The algorithm prioritizes nodes with lower heuristic values, believing they are closer to the goal.

Step-by-Step Algorithm for Greedy Best-First Search:

Step 1. Initialize the Priority Queue and Visited Set:

- Create an empty priority queue to store nodes ordered by their heuristic values.
- Create an empty set to keep track of visited nodes.

Step 2. Enqueue the Start Node with its Heuristic Value:

- Begin the Greedy Best-First Search by enqueueing the starting node (initial state) into the priority queue along with its heuristic value $h(\text{start})$.
- Mark the starting node as visited by adding it to the visited set.

Step 3. Repeat Until the Priority Queue is Empty:

- While the priority queue is not empty, do the following steps:

Step 4. Dequeue a Node from the Priority Queue:

- Remove the node with the lowest heuristic value from the priority queue. This node will be the current node that we are exploring.

Step 5. Process the Current Node:

- Examine the current node to determine if it is the goal state or if it has any unvisited neighbors that need to be explored.

Step 6. Explore Unvisited Neighbors with their Heuristic Values:

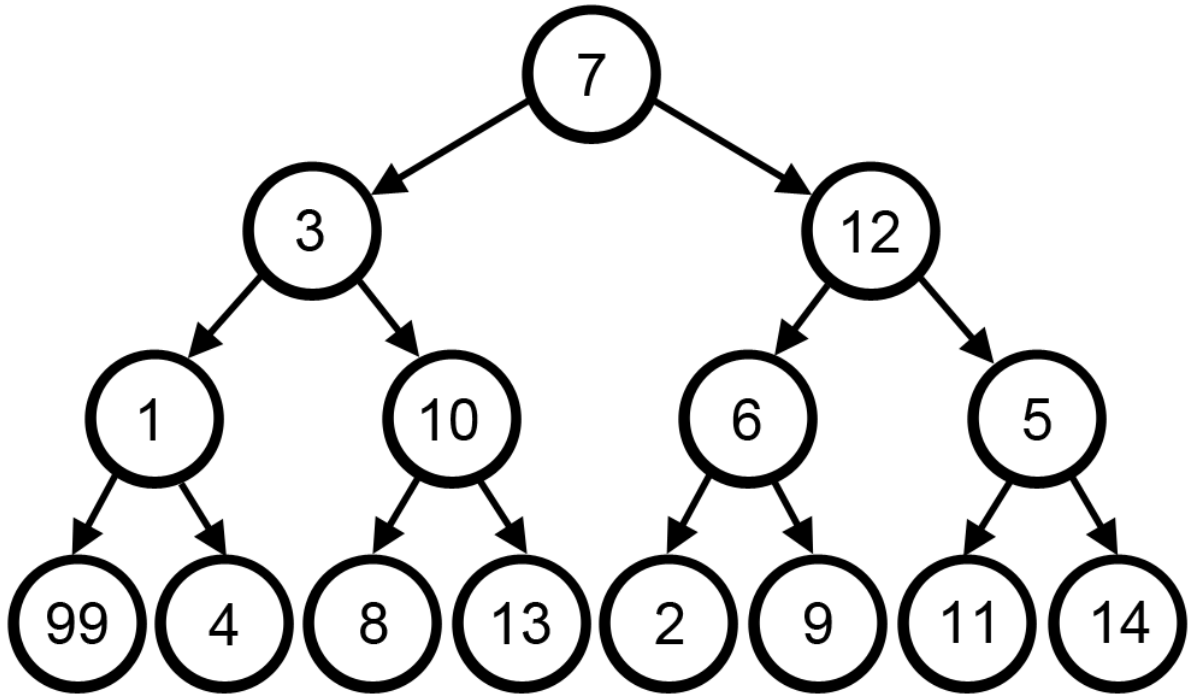
- For each unvisited neighbor of the current node, calculate its heuristic value $h(\text{neighbor})$ using the heuristic function.
- Enqueue the neighbor into the priority queue along with its heuristic value $h(\text{neighbor})$.
- Mark the neighbor as visited by adding it to the visited set.

Step 7. Repeat the Process:

- Repeat **steps 4 to 6** until the priority queue becomes empty, which means either the goal state is found or all possible paths have been explored.

Heuristic functions are essential components of Greedy Best-First Search. These functions provide an estimation of the remaining cost or distance to reach the goal state from a given node. The quality of the heuristic function influences the algorithm's effectiveness in prioritizing nodes correctly, guiding it towards promising regions of the problem space. A good heuristic function should be admissible, meaning it never overestimates the actual cost to reach the goal. Admissible heuristics guarantee that the Greedy Best-First Search will always find the optimal solution if one exists.

Vizulization (resource <https://commons.wikimedia.org/wiki/File:Greedy-search-path.gif>)



Let's explore a simple Python implementation of Greedy Best-First Search to find the shortest path in a graph with weighted edges. We will use the Manhattan distance heuristic for illustrative purposes:

```

import heapq

def heuristic_manhattan(node, goal):
    node_x, node_y = node
    goal_x, goal_y = goal
    return abs(node_x - goal_x) + abs(node_y - goal_y)

def greedy_best_first_search(graph, start, goal):
    priority_queue = [(0, start)] # Priority queue with heuristic
    value as priority
    visited = set()

    while priority_queue:
        _, current_node = heapq.heappop(priority_queue)

        if current_node == goal:
            return True # Goal state is reachable

        visited.add(current_node)

        for neighbor, _ in graph[current_node]: # Ignore edge
            weights for Greedy BFS
            if neighbor not in visited:
                priority = heuristic_manhattan(neighbor, goal)
                heapq.heappush(priority_queue, (priority, neighbor))

    return False # Goal state is not reachable
  
```



```
# Example Graph with 2D coordinates as nodes:
graph = {
    (0, 0): [((1, 0), 5), ((0, 1), 3)],
    (1, 0): [((2, 0), 2), ((1, 1), 4)],
    (0, 1): [((0, 2), 7)],
    (2, 0): [],
    (1, 1): [((0, 2), 1)],
    (0, 2): [],
    (2, 1): []
}

start_node = (0, 0)
goal_node = (2, 1)

result = greedy_best_first_search(graph, start_node, goal_node)

if result:
    print("Goal state is reachable using Greedy Best-First Search.")
else:
    print("Goal state is not reachable using Greedy Best-First Search.")

Goal state is not reachable using Greedy Best-First Search.
```

In this example, we define a simple graph where nodes are represented as 2D coordinates (x, y) in a grid. The `heuristic_manhattan` function calculates the Manhattan distance between two nodes in the 2D grid. The `greedy_best_first_search` function applies Greedy Best-First Search by using a priority queue, where the priority is the heuristic value (Manhattan distance) of each node. The search explores nodes with lower heuristic values first, moving closer to the goal state.

Advantages of Greedy Best-First Search:

1. **Efficiency:** Greedy Best-First Search is generally more efficient than uninformed search algorithms like BFS and DFS due to its intelligent node prioritization based on heuristics.
2. **Quick Solutions:** In certain problem spaces, Greedy Best-First Search can provide quick but not necessarily optimal solutions,

making it suitable for scenarios where finding any feasible solution rapidly is essential.

1. **Space Efficiency:** Greedy Best-First Search requires relatively less memory compared to algorithms like BFS because it only keeps track of nodes in the priority queue.

Limitations of Greedy Best-First Search:

1. **Lack of Optimality:** Greedy Best-First Search does not guarantee finding the optimal solution in all cases. The chosen heuristic may lead the algorithm towards a suboptimal path.
2. **Completeness:** Greedy Best-First Search may not be complete in infinite state spaces or problems with no feasible solutions, as it does not explore all possibilities systematically.

3. **Heuristic Sensitivity:** The effectiveness of Greedy Best-First Search heavily relies on the quality of the chosen heuristic function. Poor heuristics may lead to inefficient or incorrect results.

Greedy Best-First Search offers valuable advantages, such as speed and space efficiency, but it comes with limitations, mainly related to optimality and heuristic quality. As we apply Greedy Best-First Search in various AI applications, it is essential to select appropriate heuristics and understand their implications to achieve desired outcomes effectively.

3.2. A* Search Algorithm - Efficient Pathfinding with Heuristic Optimization

A* Search Algorithm is a widely used informed algorithm that combines the benefits of both Dijkstra's algorithm and Greedy Best-First Search. By incorporating a heuristic function that estimates the cost from the start node to the goal node, **A*** efficiently explores the problem space to find the shortest path.

A* Search Algorithm aims to find the shortest path from the start node to the goal node in a graph or grid. It employs two main factors for each node: the actual cost from the start node to that node (denoted as $g(n)$), and the estimated cost from that node to the goal node (denoted as $h(n)$). The key idea behind **A*** is to prioritize nodes that have the lowest total cost, calculated as the sum of $g(n)$ and $h(n)$. By doing so, **A*** explores paths that are likely to lead to the optimal solution while avoiding unnecessary exploration.

Step-by-Step Algorithm for **A*** Search:

Step 1. Initialize the Open List and Closed List:

- Create an empty open list to store nodes to explore.
- Create an empty closed list to keep track of visited nodes.

Step 2. Enqueue the Start Node with $g(n)$ and $h(n)$:

- Begin the **A*** algorithm by enqueueing the starting node (initial state) into the open list, along with its actual cost $g(\text{start})$ (usually set to 0) and its heuristic value $h(\text{start})$ (estimated cost to the goal).

Step 3. Repeat Until the Open List is Empty or Goal is Reached:

- While the open list is not empty, do the following steps:

Step 4. Dequeue the Node with the Lowest $f(n)$ from the Open List:

- Remove the node with the lowest total cost $f(n) = g(n) + h(n)$ from the open list. This node will be the current node that we are exploring.

Step 5. Check for Goal Node:

- If the current node is the goal node, the shortest path is found. Terminate the algorithm and reconstruct the path.

Step 6. Add the Current Node to the Closed List:

- Mark the current node as visited by adding it to the closed list.

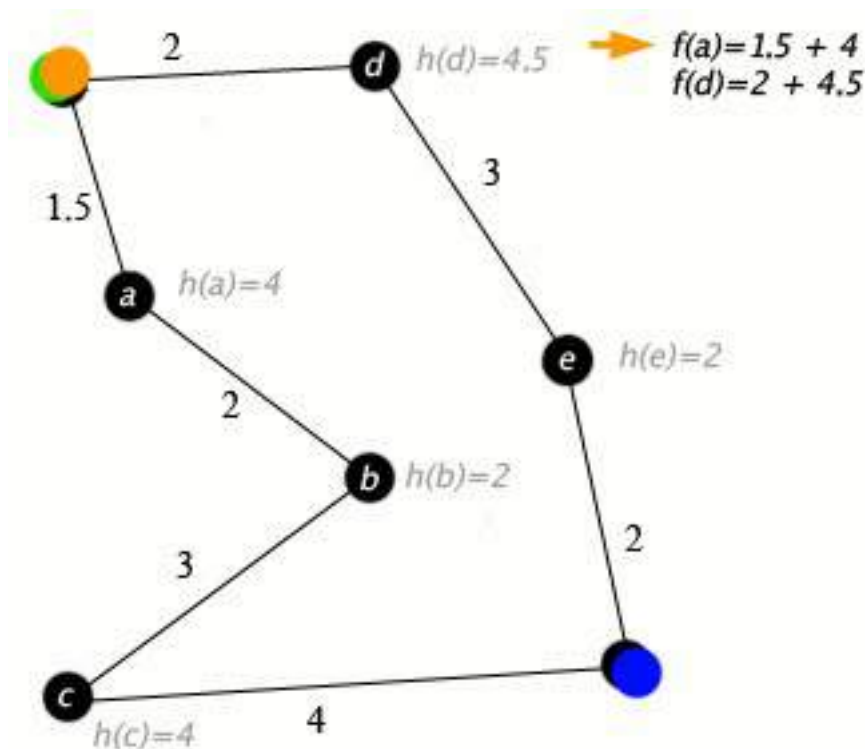
Step 7. Explore Neighbors:

- For each unvisited neighbor of the current node, calculate its actual cost $g(\text{neighbor})$ from the start node ($g(\text{current}) + \text{cost to neighbor}$).
- Calculate the heuristic value $h(\text{neighbor})$ for the neighbor, estimating its cost to reach the goal.
- If the neighbor is already in the open list with a lower f -value, skip it. Otherwise, enqueue it into the open list.

Step 8. Repeat the Process:

- Repeat steps 4 to 7 until the open list becomes empty (no path exists) or the goal node is reached.

Visualization (resource https://en.wikipedia.org/wiki/A*_search_algorithm)



Let's explore a Python implementation of A* Search to find the shortest path in a weighted graph. We'll use a 2D grid representation for simplicity:

```
import heapq

def heuristic_manhattan(node, goal):
    node_x, node_y = node
    goal_x, goal_y = goal
    return abs(node_x - goal_x) + abs(node_y - goal_y)

def a_star_search(graph, start, goal):
    open_list = [(0, start)] # Priority queue with f-value as
    # priority
    g_values = {start: 0} # Dictionary to store the actual cost
    # from start to each node
    came_from = {} # Dictionary to reconstruct the path
```

```

while open_list:
    f, current_node = heapq.heappop(open_list)

    if current_node == goal:
        # Reconstruct the path
        path = []
        while current_node in came_from:
            path.append(current_node)
            current_node = came_from[current_node]
        path.append(start)
        path.reverse()
        return path

    for neighbor, cost in graph[current_node]:
        g_value = g_values[current_node] + cost
        if neighbor not in g_values or g_value <
g_values[neighbor]:
            g_values[neighbor] = g_value
            h_value = heuristic_manhattan(neighbor, goal)
            f_value = g_value + h_value
            heapq.heappush(open_list, (f_value, neighbor))
            came_from[neighbor] = current_node

    return None # Goal state is not reachable

# Example Graph with 2D coordinates as nodes:
graph = {
    (0, 0): [((1, 0), 5), ((0, 1), 3)],
    (1, 0): [((2, 0), 2), ((1, 1), 4)],
    (0, 1): [((0, 2), 7)],
    (2, 0): [],
    (1, 1): [((0, 2), 1)],
    (0, 2): []
}

start_node = (0, 0)
goal_node = (1, 1)

path = a_star_search(graph, start_node, goal_node)

if path:
    print("Shortest Path:", path)
else:
    print("Goal state is not reachable using A* Search.")

Shortest Path: [(0, 0), (1, 0), (1, 1)]

```

In this example, we use a simple graph with 2D coordinates as nodes and weighted edges. The `heuristic_manhattan` function calculates the Manhattan distance between two nodes in the 2D grid. The `a_star_search` function applies A* Search to find the shortest path from the `start_node` to the `goal_node`. If a path exists, the function returns the sequence of nodes representing the shortest path. Otherwise, it returns `None` if the goal state is not reachable.

An **admissible heuristic function** is one that never overestimates the cost to reach the goal from any node. Formally, for all nodes n , the heuristic value $h(n)$ should be less than or equal to the actual cost from n to the goal. Admissible heuristics guarantee the optimality of A^* , ensuring that it finds the shortest path if one exists.

A consistent (or monotonic) heuristic function, also known as the triangle inequality, satisfies an additional condition. For all nodes n and their successors s , the heuristic value $h(n)$ should be less than or equal to the sum of the cost from n to s and $h(s)$. Consistent heuristics ensure that the estimated cost from n to the goal is always less than or equal to the cost of moving from n to s and the estimated cost from s to the goal. All consistent heuristics are also admissible.

A^* is both optimal and complete when used with an admissible heuristic function. Optimality guarantees that A^* will always find the shortest path in a graph with non-negative edge costs, as it explores nodes with the lowest f -values first. Completeness ensures that A^* will find a solution if one exists, provided that the search space is finite. However, A^* may not be complete in infinite state spaces or graphs with cycles, as it may get stuck in an infinite loop.

3.3. Comparison between Greedy Best-First and A^*

Both Greedy Best-First Search and A^* Search are informed search algorithms that utilize heuristic functions to guide their exploration of the problem space. Despite their similarities, they exhibit significant differences in terms of completeness, optimality, and efficiency. Let's compare the two algorithms based on key aspects:

Aspect / Algorithm	Greedy Best-First Search	A^* Search
Completeness	Greedy Best-First Search is not complete, meaning it does not guarantee finding a solution even if one exists. It can get stuck in infinite loops or fail to explore all possible paths.	A^* Search is complete when used with an admissible heuristic. It will always find a solution if one exists, provided that the search space is finite.
Optimality	Greedy Best-First Search does not guarantee finding the optimal solution. It tends to prioritize nodes based solely on their heuristic values, which may lead to suboptimal paths.	A^* Search guarantees optimality when used with an admissible heuristic. It explores paths with the lowest total cost (actual cost plus heuristic value), ensuring the shortest path is found.
Heuristic	Greedy Best-First Search relies solely on the heuristic function to prioritize nodes. It has no knowledge of the actual cost from the start node, which may result in an overly optimistic or pessimistic assessment.	A^* Search incorporates both actual cost ($g(n)$) and heuristic value ($h(n)$) into its evaluation. This combination allows A^* to strike a balance between exploration and exploitation, making more informed decisions.
Efficiency	Greedy Best-First Search is	A^* Search is efficient in

Aspect / Algorithm	Greedy Best-First Search	A* Search
	generally more efficient than uninformed search algorithms like BFS and DFS, as it uses heuristic guidance. However, it may not be as efficient as <i>A*</i> in terms of finding the optimal path.	finding the optimal path because it intelligently explores the most promising paths first based on the sum of actual cost and heuristic value. With a good admissible heuristic, A* can often reach the goal faster than other algorithms.
Memory Usage	Greedy Best-First Search requires less memory than algorithms like BFS because it only keeps track of nodes in the priority queue. However, it may still consume considerable memory depending on the size of the search space.	<i>A* Search</i> can consume significant memory due to the need to store information about the nodes in both the open list and closed list. The memory requirements can grow substantially for larger search spaces.
Applicability	Greedy Best-First Search is suitable for scenarios where a quick but not necessarily optimal solution is acceptable. It can be used in pathfinding, route planning, and heuristic optimization problems.	<i>A* Search</i> is ideal for scenarios where finding the optimal solution is crucial. It is commonly used in pathfinding, robotics, game AI, and navigation systems.

In summary, Greedy Best-First Search and *A* Search* are both informed search algorithms that utilize heuristic functions to guide their exploration. While Greedy Best-First Search is more efficient than uninformed search algorithms, it lacks completeness and optimality guarantees. *A* Search*, on the other hand, is both complete and optimal when used with an admissible heuristic, making it a preferred choice for problems where finding the shortest path or optimal solution is essential. However, the choice between the two algorithms depends on the specific requirements and constraints of the problem at hand.

4. Local Search Algorithms (Duration: 1 hour)

In the pursuit of solving complex optimization and search problems, Local Search Algorithms offer a versatile and powerful approach. Unlike global search methods that explore the entire search space, local search algorithms focus on iteratively improving a given solution by exploring neighboring states. By making small incremental changes and evaluating their impact on the objective function, local search algorithms aim to find the optimal or near-optimal solution within a restricted region of the search space. These algorithms are particularly well-suited for large-scale problems with a vast number of possible solutions. In this session, we will delve into the concept of Local Search Algorithms, explore popular techniques such as Hill Climbing, Simulated Annealing, and Genetic Algorithms, and understand their strengths and limitations in tackling a wide range of real-world optimization challenges.

4.1. Hill-Climbing Algorithm - Navigating Local Peaks in Search of the Summit

Hill-Climbing is a popular local search technique employed in optimization problems, where the goal is to find the best possible solution within a restricted neighborhood of the search space. Inspired by the concept of climbing a hill, this algorithm iteratively explores neighboring solutions and moves towards higher-elevation states until reaching a peak, a local optimum.

Step-by-Step Explanation of Hill-Climbing Algorithm:

Step 1. Initialize the Current State:

- Begin the Hill-Climbing algorithm by initializing the current state as a random or user-defined starting point within the search space.

Step 2. Evaluate the Current State:

- Calculate the objective function value for the current state, representing the quality of the solution.

Step 3. Repeat Until No Better Neighbor:

- While there exist neighboring states with better objective function values than the current state, do the following steps:

Step 4. Explore Neighboring States:

- Generate neighboring states by making small modifications or perturbations to the current state. These modifications can be incremental changes or random mutations.

Step 5. Evaluate Neighbors:

- Calculate the objective function values for each neighboring state.

Step 6. Move to the Best Neighbor:

- Move to the neighboring state with the best objective function value. If multiple neighbors have the same or similar values, the algorithm may choose one randomly.

Step 7. Repeat or Terminate:

- Repeat **steps 3 to 6** until the current state has no better neighbors, indicating that a local optimum is reached, or until a specified termination condition is met.

Types of Hill-Climbing:

1. *Simple Hill-Climbing* In Simple Hill-Climbing, the algorithm always moves to the first neighboring state with a better objective function value. If no such neighbor is found, the algorithm terminates, even if the current state is not a local optimum.
2. *Steepest-Ascent Hill-Climbing* Steepest-Ascent Hill-Climbing, also known as steepest ascent or steepest ascent climbing, evaluates all neighboring states and moves to the state with the best objective function value. It chooses the steepest upward slope, making it less likely to get stuck in local optima.

Let's explore a Python implementation of Simple Hill-Climbing to find the maximum value of a simple function:

```
import random

def objective_function(x):
    # Example objective function, modify according to your problem
    return -(x ** 2) + 10 * x + 25

def hill_climbing(initial_state, step_size, max_iterations):
    current_state = initial_state

    for _ in range(max_iterations):
        current_value = objective_function(current_state)
        neighbor_state = current_state + random.uniform(-step_size,
step_size)
        neighbor_value = objective_function(neighbor_state)

        if neighbor_value > current_value:
            current_state = neighbor_state

    return current_state

# Example Usage:
initial_state = random.uniform(-10, 10)
step_size = 0.1
max_iterations = 100

optimal_solution = hill_climbing(initial_state, step_size,
max_iterations)
print("Optimal Solution:", optimal_solution)

Optimal Solution: -7.076726171697445
```

In this example, we use a simple function `objective_function(x)` with a single variable `x`. The Hill-Climbing algorithm starts from a randomly chosen initial state and iteratively moves to better neighboring states until a local optimum is reached or the maximum number of iterations is reached.

Hill-Climbing algorithms are prone to getting stuck in *local optima*, where they cannot find better solutions due to the surrounding states having lower objective function values. Additionally, they may encounter *plateaus*, where a large number of neighboring states have similar or equal objective function values, making it challenging to find an upward direction.

Hill-Climbing Algorithm is a powerful local search technique used in optimization problems. By iteratively exploring neighboring solutions and moving towards higher-elevation states, the algorithm seeks to find local optima within a restricted region of the search space. While Hill-Climbing can be effective for certain problems, it faces challenges when dealing with plateaus and getting trapped in local optima. As we apply Hill-Climbing and its variations, it is essential to consider the choice of starting point, step size, and termination conditions to enhance its efficiency and effectiveness in finding optimal solutions.

###4.2. Simulated Annealing

Simulated Annealing is a powerful probabilistic optimization technique inspired by the annealing process in metallurgy. This algorithm mimics the gradual cooling of a material to minimize defects and reach an optimal state. Similarly, Simulated Annealing seeks to find the global optimum of a problem by allowing controlled random movements, which enable it to escape local optima and explore a broader search space.

Step-by-Step Explanation of Simulated Annealing Algorithm:

Step 1. Initialize the Current State:

- Begin the Simulated Annealing algorithm by initializing the current state as a random or user-defined starting point within the search space.

Step 2. Set the Initial Temperature:

- Define the initial temperature, which controls the extent of randomness in the exploration. A higher initial temperature allows more random movements, aiding in escaping local optima initially.

Step 3. Repeat Until Temperature is Low:

- While the temperature is above a predefined minimum value (close to zero), do the following steps:

Step 4. Explore Neighbor:

- Generate a neighboring state by making a small random perturbation to the current state.

Step 5. Evaluate the Change in Objective Function:

- Calculate the difference in the objective function value between the current state and the neighboring state (ΔE).

Step 6. Acceptance Probability:

- Calculate the probability of accepting the neighboring state using the acceptance probability formula. This formula incorporates the temperature, ΔE , and a Boltzmann constant, controlling the probability of accepting worse solutions. As the temperature decreases, the probability of accepting worse solutions diminishes, leading to a more focused search.

Step 7. Accept or Reject the Neighbor:

- Compare the acceptance probability to a random number between 0 and 1. If the acceptance probability is greater than the random number, accept the neighboring state as the new current state. Otherwise, reject it and stay in the current state.

Step 8. Cooling Schedule:

- Update the temperature according to the cooling schedule, which gradually decreases the temperature over time. Common cooling schedules include exponential, linear, or logarithmic cooling.

Step 9. Repeat or Terminate:

- Repeat steps 4 to 8 until the temperature reaches the minimum value or a termination condition is met.

Below is a Python implementation of the Simulated Annealing algorithm to find the minimum value of a simple function:

```
import math
import random

def objective_function(x):
    # Example objective function, modify according to your problem
    return x ** 2 - 10 * x + 25

def acceptance_probability(delta_E, temperature):
    if delta_E < 0:
        return 1.0
    return math.exp(-delta_E / temperature)

def simulated_annealing(initial_state, initial_temperature,
                        cooling_rate, min_temperature, max_iterations):
    current_state = initial_state
    current_value = objective_function(current_state)
    best_state = current_state
    best_value = current_value
    temperature = initial_temperature

    for _ in range(max_iterations):
        neighbor_state = current_state + random.uniform(-1, 1) #
        # Random perturbation

        neighbor_value = objective_function(neighbor_state)
        delta_E = neighbor_value - current_value

        if acceptance_probability(delta_E, temperature) >
        random.uniform(0, 1):
            current_state = neighbor_state
            current_value = neighbor_value

        if current_value < best_value:
            best_state = current_state
            best_value = current_value

        temperature *= cooling_rate
        if temperature < min_temperature:
            break

    return best_state

# Example Usage:
initial_state = random.uniform(-10, 10)
initial_temperature = 100.0
cooling_rate = 0.95
min_temperature = 0.1
max_iterations = 1000

optimal_solution = simulated_annealing(initial_state,
                                       initial_temperature, cooling_rate, min_temperature, max_iterations)
```

```
print("Optimal Solution:", optimal_solution)
print("Minimum Value:", objective_function(optimal_solution))
```

```
Optimal Solution: 4.991082678833491
Minimum Value: 7.951861678279215e-05
```

In this example, we use a simple function `objective_function(x)` with a single variable `x`. The Simulated Annealing algorithm starts from a randomly chosen initial state and iteratively explores neighboring states to find the global minimum of the function.

Parameters:

- `initial_state`: The initial state or starting point in the search space.
- `initial_temperature`: The initial temperature to control the exploration at the beginning of the algorithm.
- `cooling_rate`: The rate at which the temperature decreases during the cooling schedule.
- `min_temperature`: The minimum temperature at which the algorithm terminates.
- `max_iterations`: The maximum number of iterations or steps allowed for the algorithm.

The algorithm uses the `acceptance_probability` function to determine whether to accept a neighboring state with a worse objective function value or not. The probability of accepting a worse solution decreases as the temperature decreases.

The algorithm gradually cools down (reduces the temperature) following the cooling schedule and explores the search space, accepting moves that improve or even worsen the objective function value based on the acceptance probability. As the temperature decreases, the algorithm becomes more selective, converging towards the global minimum.

Note: In real-world applications, the Simulated Annealing algorithm may require tuning of the parameters (initial temperature, cooling rate, etc.) based on the specific problem at hand to achieve the desired results effectively.

The *annealing schedule* controls the temperature reduction rate, determining how quickly the algorithm shifts from exploration to exploitation. A slow cooling schedule allows more exploration at higher temperatures, while a faster schedule focuses on exploitation as the temperature decreases.

The *acceptance probability* is based on the Boltzmann distribution and determines the likelihood of accepting a worse solution. It ensures a controlled balance between exploration and exploitation during the search. A high acceptance probability at higher temperatures allows the algorithm to explore the search space, while a low acceptance probability at lower temperatures enables exploitation and convergence to the optimal solution.

Simulated Annealing overcomes local optima through the *stochastic acceptance* of worse solutions. At higher temperatures, the algorithm has a higher probability of accepting worse solutions, allowing it to escape local optima and explore new regions. As the temperature decreases, the algorithm becomes more selective, favoring better solutions and gradually converging towards the global optimum.

Simulated Annealing Algorithm is a versatile optimization technique that employs controlled randomness to navigate the search space and find near-optimal solutions. By utilizing annealing schedules and acceptance probabilities, it balances exploration and exploitation,

enabling it to overcome local optima effectively. Simulated Annealing is widely applicable to a range of optimization problems, from function optimization to combinatorial optimization and parameter tuning. As we apply Simulated Annealing in various domains, understanding the influence of temperature and cooling schedules will enable us to fine-tune the algorithm for improved performance and efficient global optimization.

5. Real-world Applications and Case Studies (Duration: 30 minutes)

N-Queens problem

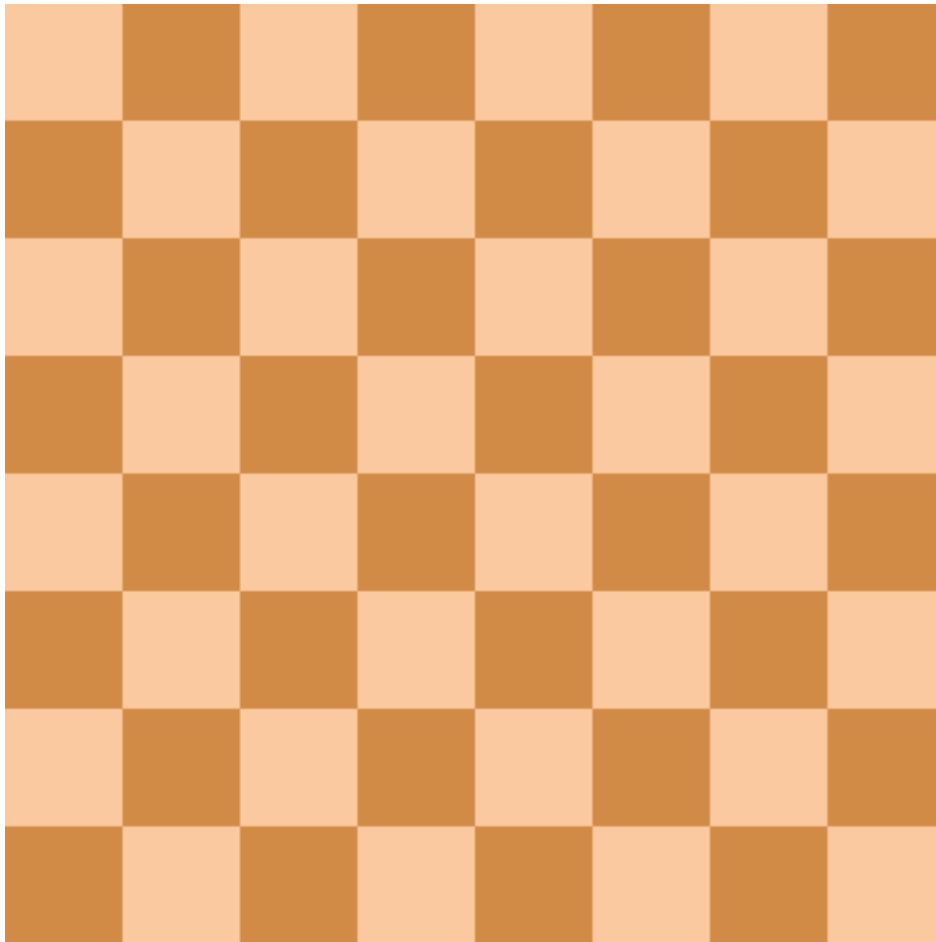
The N-Queens problem is a classic puzzle that involves placing N chess queens on an $N \times N$ chessboard so that no two queens threaten each other. In other words, no two queens can share the same row, column, or diagonal. The objective of the problem is to find the configuration in which N queens can be placed on the board satisfying these constraints.

The name "N-Queens" comes from the fact that the problem is often described and studied for a generalized case where N queens are to be placed on an $N \times N$ board. However, the problem can be solved for smaller board sizes, such as the traditional 8×8 chessboard.

The N-Queens problem is known to be a combinatorial optimization problem and is considered to be a well-known example of the backtracking algorithm. Backtracking involves systematically trying out different possibilities and undoing choices when they are found to be invalid.

The N-Queens problem has captured the interest of mathematicians and computer scientists due to its challenging nature and application in various fields, including computer science, artificial intelligence, and algorithm analysis. Solving the N-Queens problem efficiently for large values of N remains an active area of research.

Visualization (resource <https://commons.wikimedia.org/wiki/File:Eight-queens-animation.gif>)



```
def solve_n_queens(n):
    # heuristic function
    def heuristic(state):
        conflicts = 0
        for i in range(len(state)):
            for j in range(i + 1, len(state)):
                if state[i] == state[j] or abs(state[i] - state[j])
== j - i:
                    conflicts += 1
        return conflicts

    def is_goal(state):
        return len(set(state)) == len(state)

    def generate_successors(state):
        successors = []
        for col in range(n):
            for row in range(n):
                if state[col] != row:
                    is_diagonal = False
                    for k in range(len(state)):
                        if abs(state[k] - row) == abs(k - col):
                            is_diagonal = True
                            break
                    if not is_diagonal:
```

```

        new_state = list(state)
        new_state[col] = row
        if tuple(new_state) not in closed_list: #
Check if the state is already visited
            successors.append((new_state,
heuristic(new_state)))
        return successors

start_state = [0] * n
open_list = [(start_state, heuristic(start_state))]
closed_list = set()

while open_list:
    open_list.sort(key=lambda x: x[1]) # Sort open list based
on heuristic value
    current_state, _ = open_list.pop(0)

    if is_goal(current_state):
        return current_state

    closed_list.add(tuple(current_state))

    successors = generate_successors(current_state)
    for successor_state, successor_heuristic in successors:
        open_list.append((successor_state, successor_heuristic))

return None

def print_solution(solution):
    if solution is None:
        print("No solution found.")
    else:
        for i in range(len(solution)):
            row = ['#'] * len(solution)
            row[solution[i]] = 'Q'
            print(' '.join(row))

n = 8 # Number of queens
print("Greedy Best-First Search:")
a_star_solution = solve_n_queens(n)
print_solution(a_star_solution)

Greedy Best-First Search:
# # Q # # # # #
# # # # # # # Q
# # # Q # # # #
# # # # # # Q #
Q # # # # # # #
# # # # # Q # #
# Q # # # # # #
# # # # Q # # #

```

For our task, it was irrelevant to use the A*search approach, because we don't have a cost for the queen's placement (or we do, but it's the same at each step).

Connect four

Four in Line, also known as *Connect Four*, is a two-player board game where the objective is to be the first to create a line of four of your tokens in any direction: horizontally, vertically, or diagonally. The game is typically played on a 7x6 grid, and each player takes turns dropping their colored tokens (X and Y) into one of the columns. The tokens will occupy the lowest available position within the chosen column. The game continues until one player successfully forms a line of four tokens or until the board is completely filled, resulting in a tie.

```
import numpy as np

# Constants
ROWS = 6
COLS = 7
EMPTY = "0"
PLAYER1 = "X"
PLAYER2 = "Y"
DIFFICULTY = 5
WIN_SCORE = 1000000

def create_board():
    return np.full((ROWS, COLS), EMPTY)

def is_valid_move(board, col):
    return board[0][col] == EMPTY

def make_move(board, col, player):
    for row in range(ROWS-1, -1, -1):
        if board[row][col] == EMPTY:
            board[row][col] = player
            break

def get_winner(board):
    # Check rows
    for row in range(ROWS):
        for col in range(COLS - 3):
            if (
                board[row][col] != EMPTY
                and board[row][col] == board[row][col + 1]
                == board[row][col + 2] == board[row][col + 3]
            ):
                return board[row][col]

    # Check columns
    for col in range(COLS):
        for row in range(ROWS - 3):
```

```

        if (
            board[row][col] != EMPTY
            and board[row][col] == board[row + 1][col]
            == board[row + 2][col] == board[row + 3][col]
        ):
            return board[row][col]

    # Check diagonal (top-left to bottom-right)
    for row in range(ROWS - 3):
        for col in range(COLS - 3):
            if (
                board[row][col] != EMPTY
                and board[row][col] == board[row + 1][col + 1]
                == board[row + 2][col + 2] == board[row + 3][col +
3]
            ):
                return board[row][col]

    # Check diagonal (bottom-left to top-right)
    for row in range(3, ROWS):
        for col in range(COLS - 3):
            if (
                board[row][col] != EMPTY
                and board[row][col] == board[row - 1][col + 1]
                == board[row - 2][col + 2] == board[row - 3][col +
3]
            ):
                return board[row][col]

    return None

def evaluate_position(board, player):
    opponent = PLAYER2 if player == PLAYER1 else PLAYER1

    # Check if the player has won
    if get_winner(board) == player:
        return WIN_SCORE

    # Check if the opponent has won
    if get_winner(board) == opponent:
        return -WIN_SCORE

    # Count the number of player's and opponent's pieces in each
    window
    score = 0

    # Evaluate rows
    for row in range(ROWS):
        for col in range(COLS - 3):
            window = board[row, col:col+4]
            score += evaluate_window(window, player, opponent)

    # Evaluate columns

```



```

    for col in range(COLS):
        for row in range(ROWS - 3):
            window = board[row:row+4, col]
            score += evaluate_window(window, player, opponent)

    # Evaluate diagonal (top-left to bottom-right)
    for row in range(ROWS - 3):
        for col in range(COLS - 3):
            window = [board[row+i][col+i] for i in range(4)]
            score += evaluate_window(window, player, opponent)

    # Evaluate diagonal (bottom-left to top-right)
    for row in range(3, ROWS):
        for col in range(COLS - 3):
            window = [board[row-i][col+i] for i in range(4)]
            score += evaluate_window(window, player, opponent)

    return score

def evaluate_window(window, player, opponent):
    # Scoring could be improved by changing the scores for different
    # positions
    score = 0
    player_count = np.count_nonzero(window == player)
    opponent_count = np.count_nonzero(window == opponent)
    empty_count = np.count_nonzero(window == EMPTY)

    if player_count == 4:
        score += 1000
    elif player_count == 3 and empty_count == 1:
        score += 5
    elif player_count == 2 and empty_count == 2:
        score += 2

    if opponent_count == 3 and empty_count == 1:
        score -= 4

    return score

def get_valid_moves(board):
    return [col for col in range(COLS) if is_valid_move(board, col)]

def is_terminal_state(board):
    return get_winner(board) is not None or
    len(get_valid_moves(board)) == 0

# Implements the minimax algorithm with alpha-beta pruning
def minimax(board, depth, alpha, beta, maximizing_player):
    valid_moves = get_valid_moves(board)

    if depth == 0 or is_terminal_state(board):
        return evaluate_position(board, PLAYER1)

```

```

if maximizing_player:
    max_score = float("-inf")
    for move in valid_moves:
        next_board = board.copy()
        make_move(next_board, move, PLAYER1)
        score = minimax(next_board, depth-1, alpha, beta, False)
        max_score = max(max_score, score)
        alpha = max(alpha, score)
        if alpha >= beta:
            break
    return max_score
else:
    min_score = float("inf")
    for move in valid_moves:
        next_board = board.copy()
        make_move(next_board, move, PLAYER2)
        score = minimax(next_board, depth-1, alpha, beta, True)
        min_score = min(min_score, score)
        beta = min(beta, score)
        if alpha >= beta:
            break
    return min_score

def make_ai_move(board):
    valid_moves = get_valid_moves(board)
    best_score = float("-inf")
    best_move = valid_moves[0]

    for move in valid_moves:
        next_board = board.copy()
        make_move(next_board, move, PLAYER1)
        score = minimax(next_board, depth=DIFFICULTY, alpha=float("-inf"), beta=float("inf"), maximizing_player=False)

        if score > best_score:
            best_score = score
            best_move = move

    make_move(board, best_move, PLAYER1)
    print("AI Player (X) made a move in column", best_move + 1)

def make_human_move(board):
    while True:
        col = int(input("Your turn! Enter the column number (1-7):")) - 1
        if col in get_valid_moves(board):
            make_move(board, col, PLAYER2)
            break
        else:
            print("Invalid move. Please try again.")

```

```

def print_board(board):
    print("Connect Four Board:")
    print("-----")
    print(board)
    print("-----")

def play_connect_four():
    board = create_board()
    print("Welcome to Connect Four!")

    while not is_terminal_state(board):
        print_board(board)
        make_ai_move(board)
        if is_terminal_state(board):
            break
        print_board(board)
        make_human_move(board)

    print_board(board)
    winner = get_winner(board)
    if winner:
        print("Game Over! Winner: ", winner)
    else:
        print("Game Over! It's a draw.")

```

Start the game

```
play_connect_four()
```

Welcome to Connect Four!

Connect Four Board:

```

-----
[['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']]
-----

```

AI Player (X) made a move in column 1

Connect Four Board:

```

-----
[['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['X' '0' '0' '0' '0' '0' '0']]
-----

```

Your turn! Enter the column number (1-7): 0

Invalid move. Please try again.

Your turn! Enter the column number (1-7): 2

Connect Four Board:

```
-----
```

```

[['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['X' 'Y' '0' '0' '0' '0' '0']]

```

 AI Player (X) made a move in column 1
 Connect Four Board:

```

[['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['X' '0' '0' '0' '0' '0' '0']
 ['X' 'Y' '0' '0' '0' '0' '0']]

```

 Your turn! Enter the column number (1-7): 4
 Connect Four Board:

```

[['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['X' '0' '0' '0' '0' '0' '0']
 ['X' 'Y' '0' 'Y' '0' '0' '0']]

```

 AI Player (X) made a move in column 1
 Connect Four Board:

```

[['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['X' '0' '0' '0' '0' '0' '0']
 ['X' '0' '0' '0' '0' '0' '0']
 ['X' 'Y' '0' 'Y' '0' '0' '0']]

```

 Your turn! Enter the column number (1-7): 3
 Connect Four Board:

```

[['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['X' '0' '0' '0' '0' '0' '0']
 ['X' '0' '0' '0' '0' '0' '0']
 ['X' 'Y' 'Y' 'Y' '0' '0' '0']]

```

 AI Player (X) made a move in column 1
 Connect Four Board:

```

[['0' '0' '0' '0' '0' '0' '0']
 ['0' '0' '0' '0' '0' '0' '0']
 ['X' '0' '0' '0' '0' '0' '0']]

```

```
['X' '0' '0' '0' '0' '0' '0']  
['X' '0' '0' '0' '0' '0' '0']  
['X' 'Y' 'Y' 'Y' '0' '0' '0']]
```

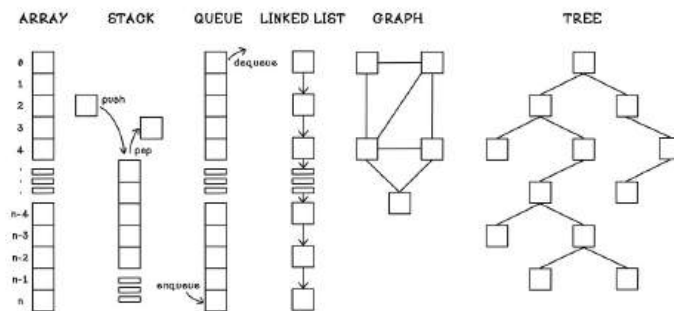
Game Over! Winner: X

6. Q&A and Wrap-up (Duration: 10 minutes)

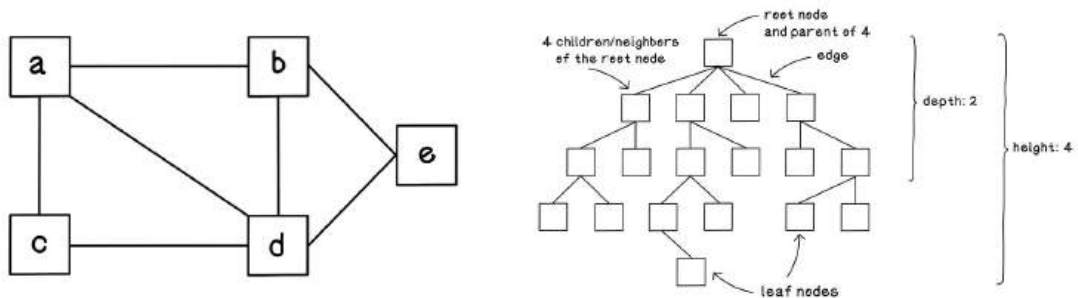
In conclusion, Search Algorithms play a vital role in various artificial intelligence and optimization applications. They enable AI systems to navigate complex problem spaces, find optimal solutions, and make informed decisions efficiently. Uninformed search algorithms like Depth-First Search and Breadth-First Search are effective for smaller search spaces, while informed search algorithms like A* and Simulated Annealing provide enhanced efficiency and optimality by leveraging heuristic knowledge.

Data structures are important to solve problems.

Search algorithms are useful in planning and finding solutions in some changing environments.

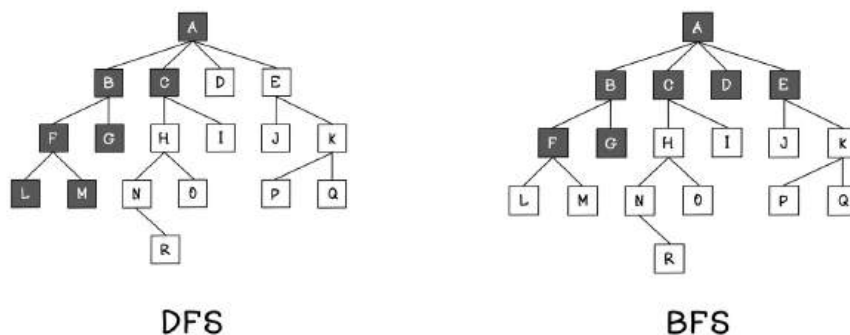


Graph and tree data structures are useful in AI.



Uninformed search is blind and can be computationally expensive. Using the correct data structures helps.

Depth-first search looks deep before looking wide. Breadth-first search looks wide before looking deep.



However, these algorithms also face challenges, such as getting trapped in local optima or dealing with the vastness of search spaces. Choosing the right search algorithm depends on the specific problem requirements, the size of the search space, and the need for optimality

or quick solutions. By understanding and leveraging the strengths of different search algorithms, AI practitioners can develop sophisticated and effective solutions for a wide range of real-world problems.

Search Algorithms in AI. Practical Tasks

Task 1. Word Search Puzzle Solver

You are tasked with implementing a Word Search Puzzle Solver in Python. The goal is to find a list of words within a 2D grid of letters. Words can be arranged horizontally, vertically, or diagonally, in either forward or reverse order. You need to create two functions to accomplish this task:

1. **search_word(grid, word)**
 - This function is responsible for searching for a specific word within a 2D grid of letters.
 - It checks all possible directions (horizontal, vertical, and diagonal) starting from each cell in the grid.
 - If the word is found in any direction, it returns **True**; otherwise, it returns **False**.
 - This function is called for each word to be searched in the grid.

Parameters:

- **grid** (list of list of str): A 2D grid of letters.
- **word** (str): The word to search for.

Returns:

- **bool**: **True** if the word is found in the grid, **False** otherwise.
1. **find_words(grid, words)**
 - This function is responsible for finding a list of words within a 2D grid of letters by calling the **search_word** function.
 - It iterates through the list of words and checks each word in the grid.
 - If a word is found, it is added to the list of found words.
 - The function returns a list of words that are found within the grid.

Parameters:

- **grid** (list of list of str): A 2D grid of letters.
- **words** (list of str): A list of words to find in the grid.

Returns:

- **list of str**: A list of words found in the grid.

This **Word Search Puzzle Solver** is designed to locate words within a grid of letters and provide a list of found words as the output.

```
!pip install git+https://github.com/mehalyna/cooltest.git
from cooltest.test_cool_search import *
@test_search_word
def search_word(grid, word):
    """
```



```

    Search for a word within a grid of letters.

    This function searches for the given word in the grid by
    checking all possible
    directions (horizontal, vertical, and diagonal) starting from
    each cell.

    Args:
    grid (list of list of str): A 2D grid of letters.
    word (str): The word to search for.

    Returns:
    bool: True if the word is found in the grid, False otherwise.
    """
    # Define possible directions (horizontal, vertical, diagonal)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1,
1), (1, -1), (1, 1)]

    return result

@test_find_words
def find_words(grid, words):
    """
    Find words within a grid of letters.

    This function searches for words in the given grid by calling
    the search_word function
    for each word to be found.

    Args:
    grid (list of list of str): A 2D grid of letters.
    words (list of str): A list of words to find in the grid.

    Returns:
    list of str: A list of words found in the grid.
    """

    return result

# Example usage:
grid = [['A', 'B', 'C', 'D'], ['E', 'F', 'G', 'O'], ['I', 'J', 'K',
'G'], ['M', 'N', 'H', 'P']]
word_list = ["HELLO", "WORLD", "HI", "FOOD", "DOG", "GOD"]
found_words = find_words(grid, word_list)
print("Found words:", found_words)

```

Task 2. Pathfinding Basics

You are tasked with implementing a basic pathfinding algorithm in Python. The goal is to find the shortest path from a starting point to a target point on a grid. The grid represents a simple maze, where some cells are blocked and cannot be traversed. You need to implement a function that finds the shortest path using a *breadth-first search* (BFS) algorithm.

Requirements:

1. Implement a function called **find_shortest_path** that takes a 2D grid, a start point, and a target point as input and returns the shortest path as a list of coordinates.

The grid is represented as a 2D list of characters, where:

- **'S'** represents the starting point.
- **'E'** represents the target point.
- **'X'** represents blocked cells.
- **' '** (space) represents open cells that can be traversed.

The function should perform a breadth-first search (BFS) to find the shortest path from the starting point to the target point.

If there is no valid path to the target point, the function should return an empty list.

If there are multiple valid shortest paths, the function should return any of them.

1. **is_valid(cell, grid)** This function checks whether a given cell is a valid open cell within the grid. It is used to determine if a cell can be traversed or if it is blocked.
 - *Parameters:*
 - **cell** (tuple): The cell coordinates in the form of **(row, col)** to be checked.
 - **grid** (list of list of str): A 2D grid of characters representing the maze.
 - *Returns:*
 - **bool**: **True** if the cell is a valid open cell that can be traversed, **False** otherwise.
1. **get_neighbors(cell, grid)** This function retrieves neighboring cells that are valid open cells within the grid. It is used to find adjacent cells that can be explored in a pathfinding algorithm.
 - *Parameters:*
 - **cell** (tuple): The cell coordinates in the form of **(row, col)** for which neighboring cells are sought.
 - **grid** (list of list of str): A 2D grid of characters representing the maze.
 - *Returns:*
 - **list of tuple**: A list of neighboring cell coordinates in the form of tuples **(row, col)**.

These functions are designed to assist in pathfinding by determining the validity of cells within the maze grid and finding neighboring cells that can be considered for traversal.

```
from collections import deque

@test_is_valid
def is_valid(cell, grid):
    """
    Check if a cell is a valid open cell in the grid.

    Args:
    cell (tuple): The cell coordinates (row, col).
    grid (list of list of str): A 2D grid of characters.
```

```

Returns:
bool: True if the cell is a valid open cell, False otherwise.
"""

    return result

@test_get_neighbors
def get_neighbors(cell, grid):
    """
    Get neighboring cells that are valid open cells in the grid.

    Args:
    cell (tuple): The cell coordinates (row, col).
    grid (list of list of str): A 2D grid of characters.

    Returns:
    list of tuple: List of neighboring cell coordinates [(row1, col1), (row2, col2), ...].
    """

    return result

@test_find_shortest_path
def find_shortest_path(grid, start, target):
    """
    Find the shortest path from the starting point to the target point on a grid.

    This function uses a breadth-first search (BFS) algorithm to find the shortest path from the starting point to the target point on the grid. The grid is represented as a 2D list of characters, where 'S' is the starting point, 'E' is the target point, 'X' are blocked cells, and ' ' (space) are open cells.

    Args:
    grid (list of list of str): A 2D grid of characters.
    start (tuple): The starting point coordinates (row, col).
    target (tuple): The target point coordinates (row, col).

    Returns:
    list of tuple: The shortest path as a list of coordinate tuples [(row1, col1), (row2, col2), ...].
    An empty list is returned if there is no valid path.
    """

    return result

# Example usage:

```

```

grid = [
    ['S', ' ', ' ', ' ', ' ', ' '],
    ['X', 'X', ' ', ' ', 'E'],
    [' ', ' ', 'X', ' ', ' '],
    ['X', 'X', ' ', 'X', ' '],
    [' ', ' ', ' ', ' ', ' ']]

start_point = (0, 0)
end_point = (1, 4)

shortest_path = find_shortest_path(grid, start_point, end_point)
print("Shortest Path:", shortest_path)

```

Task 3. Maze Solving with Depth-First Search

You are tasked with developing a program that can solve a simple maze using the **Depth-First Search (DFS)** algorithm. The goal is to find a path from the starting point to the exit of the maze. The maze is represented as a grid, where some cells are walls (blocked), and others are open paths. The program should navigate through the maze, find a path from the starting point to the exit, and provide the solution.

Requirements:

1. Implement a function called `solve_maze(maze)` that takes a maze as input and returns the solution path as a list of coordinates.

The maze is represented as a 2D grid, where:

- `'S'` represents the starting point.
- `'E'` represents the exit point.
- `'X'` represents blocked cells (walls).
- `' '` (space) represents open paths.
- *Parameters:*
 - `maze` (list of list of str): A 2D grid representing the maze.
- *Returns:*
 - `list of tuple`: The solution path as a list of coordinate tuples `[(row1, col1), (row2, col2), ...]`. An empty list is returned if there is no valid path.
- 1. **dfs(row, col)** This function performs the Depth-First Search (DFS) algorithm to explore the maze and find a path from the current cell to the exit point 'E'.
- *Parameters:*
 - `maze` (list): list of lists which represents the maze.
 - `row` (int): The current row coordinate within the maze.
 - `col` (int): The current column coordinate within the maze.
 - `visited` (list of list of bool): A 2D grid representing visited cells in the maze. It keeps track of which cells have been visited to prevent revisiting them.

- `solution_path` (list of tuple): A list used to store coordinate tuples representing the solution path as it's discovered.
- *Returns:*
 - `bool`: `True` if a valid path is found from the current cell to 'E', `False` otherwise.

```
def dfs(maze, row, col, visited, solution_path):
    """
    Depth-First Search (DFS) algorithm to explore the maze and find
    a path from
    the current cell to the exit point 'E'.

    Args:
    maze (list): list of lists which represents the maze.
    row (int): The current row coordinate.
    col (int): The current column coordinate.
    visited (list of list of bool): A 2D grid representing visited
    cells in the maze.
    solution_path (list of tuple): A list of coordinate tuples
    representing the solution path.

    Returns:
    bool: True if a valid path is found from the current cell to
    'E', False otherwise.
    """

    return result

@test_solve_maze
def solve_maze(maze):
    """
    Solve a maze using Depth-First Search (DFS) algorithm.

    Args:
    maze (list of list of str): A 2D grid representing the maze.

    Returns:
    list of tuple: The solution path as a list of coordinate tuples
    [(row1, col1), (row2, col2), ...].
    An empty list is returned if there is no valid
    path.
    """

    # Find the starting point

    # Initialize the visited array and solution_path list

    return result
```

```
# Example usage:
maze = [
    ['S', ' ', 'X', 'X', 'E'],
    ['X', ' ', ' ', 'X', ' '],
    ['X', 'X', ' ', ' ', ' '],
    [' ', 'X', 'X', 'X', ' '],
    [' ', ' ', ' ', ' ', ' ']
]

path = solve_maze(maze)
print("Solution Path:", path)
```

Optimization in AI

Duration: 3 hours

Agenda:

1. Introduction to Optimization in AI
 - a. Definition and significance of optimization in AI
 - b. The role of optimization algorithms in AI applications
 - c. Types of optimization problems in AI
2. Unconstrained Optimization Techniques
 - a. Gradient Descent
 - Explanation of gradient descent
 - Batch vs. stochastic gradient descent
 - Learning rates and convergence
 - b. Newton's Method - Understanding Newton's method - Hessian matrix and convergence properties
 - c. Comparison between Gradient Descent and Newton's Method
1. Constrained Optimization and Linear Programming
 - a. Introduction to constrained optimization
 - b. Formulation of linear programming problems
 - c. Simplex algorithm for linear programming
 - d. Real-world examples of linear programming in AI
2. Combinatorial Optimization
3. Evolutionary Algorithms
 - a. Particle Swarm Optimization (PSO)
 - Explanation of PSO
 - Swarm behavior and particle updates
 - Advantages and limitations of PSO
 - c. Comparison between Simulated Annealing and Particle Swarm Optimization
4. Real-world Applications of Optimization in AI
 - a. AI in resource allocation and scheduling
 - c. AI in portfolio optimization and financial modeling

- d. AI in game playing and strategy development
- 5. Q&A and Wrap-up
 - a. Questions and answers
 - b. Summary of key concepts covered in the session
 - c. Encouraging further exploration of optimization techniques in AI

1. Introduction to Optimization in AI

Optimization, in the context of artificial intelligence, refers to the process of finding the best possible solution or set of solutions from a vast number of alternatives. The goal is to optimize or maximize certain objectives while considering specific constraints. It's like finding the shortest path on a map, the best parameters for a machine learning model, or the optimal route for a delivery vehicle. Optimization is at the core of AI, enabling systems to achieve efficiency and effectiveness in various tasks.

Optimization algorithms are essential tools that help AI systems reach their full potential. These algorithms provide systematic and efficient approaches to search through the vast solution space and find the optimal or near-optimal solutions. AI applications, such as machine learning, neural networks, and robotics, heavily rely on optimization algorithms to fine-tune model parameters, make accurate predictions, and improve overall performance.

There are several types of optimization problems encountered in AI. Let's briefly explore a few of them:

1. **Unconstrained Optimization:** In this type, the goal is to find the minimum or maximum of an objective function without any constraints. It involves finding the global or local optimum of a function.
2. **Constrained Optimization:** Constrained optimization deals with finding the optimal solution while considering specific constraints. These constraints limit the feasible region, making the problem more challenging.
3. **Combinatorial Optimization:** This type involves finding the best solution from a finite set of discrete possibilities. Problems like the traveling salesman or graph coloring fall into this category.
4. **Evolutionary Algorithms (EAs):** are a class of optimization algorithms inspired by the principles of natural selection and evolution. These algorithms mimic the process of biological evolution to find optimal solutions to complex problems. EAs maintain a population of candidate solutions and apply variation operators like mutation, crossover, and selection to iteratively evolve better solutions over generations. Due to their ability to handle large search spaces and non-linear optimization problems, Evolutionary Algorithms have found applications in various fields, including optimization, machine learning, and engineering design.

In conclusion, optimization is a critical aspect of artificial intelligence that allows systems to find the best possible solutions for various problems. Optimization algorithms play a central role in AI applications, aiding in achieving better performance and efficiency. Understanding the different types of optimization problems will provide a solid foundation for exploring more advanced topics in AI.

[Slides](#) for better understanding.

2. Unconstrained Optimization Techniques

Unconstrained Optimization Techniques are powerful methods used to find the optimal solution to a problem without imposing any constraints on the variables. In this context, "optimal" typically refers to finding either the maximum or minimum of an objective function. Unlike constrained optimization, where the feasible region is limited by constraints, unconstrained optimization allows the search for solutions across the entire feasible space. These techniques are widely employed in various domains, including machine learning, data science, and engineering, to fine-tune parameters, optimize cost functions, and improve overall performance. Unconstrained optimization problems often involve searching for the global or local optimum of the objective function, making these techniques fundamental in exploring the peak potential of many real-world applications.

2.1. Gradient Descent in Optimization

Gradient Descent is an iterative optimization algorithm used to find the minimum of a function. It works by taking steps in the direction of the steepest descent (negative gradient) of the function with respect to the model's parameters. By continuously updating the parameters based on the negative gradient, the algorithm converges towards the optimal solution.

Step-by-step algorithm for the Gradient Descent optimization method:

Step 1: Initialize the model parameters:

- Choose initial values for the model's parameters (weights and biases). These initial values can be random or set to some predefined values.

Step 2: Define the objective function:

- Specify the function that you want to optimize (minimize or maximize). This function is commonly known as the **objective function** or **cost function**.

Step 3: Calculate the gradient of the objective function:

- Compute the gradient (derivative) of the objective function with respect to each model parameter. The gradient points in the direction of the steepest increase of the function. For minimization, we take the negative gradient.

Step 4: Choose a learning rate (step size):

- The learning rate (often denoted by the symbol α) determines the size of the steps taken during each iteration. It is a hyperparameter that needs to be chosen carefully. Too large a learning rate may cause overshooting, while too small a learning rate may slow down convergence.

Step 5: Update the model parameters:

- Update each model parameter using the calculated gradient and the learning rate. The update formula is: $\text{parameter} = \text{parameter} - \text{learning\_rate} * \text{gradient}$

where `parameter` represents a model parameter (e.g., weight or bias), `learning_rate` is the chosen learning rate, and `gradient` is the calculated gradient of the objective function with respect to that parameter.

Step 6: Repeat the process:

- Iterate through **Steps 3 to 5** for a specified number of iterations or until convergence is reached. Convergence is typically determined based on some predefined criteria, such as the change in the objective function's value being below a threshold or a fixed number of iterations.

Step 7: End of Algorithm:

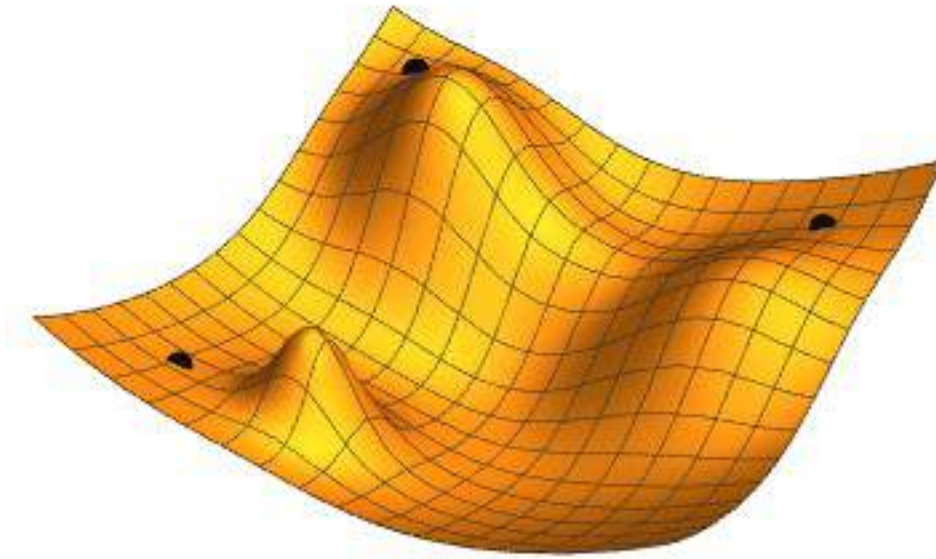
- Once the specified number of iterations is completed, the algorithm ends, and the final values of the model parameters represent the solution to the optimization problem.

It's important to note that there are variations of the basic gradient descent algorithm, such as mini-batch gradient descent and stochastic gradient descent (SGD), which involve using subsets of the data to update the model parameters. Batch gradient descent computes the gradient of the entire training dataset, making it more accurate but computationally expensive, especially for large datasets. On the other hand, SGD calculates the gradient for each training sample individually, leading to faster convergence but introducing more variance in the updates.

Additionally, there are more advanced variants like momentum, RMSprop, and Adam, which adapt the learning rate during the optimization process for faster convergence and improved performance.

Implementing and experimenting with gradient descent and its variants will provide you with a deeper understanding of optimization algorithms and their role in machine learning and artificial intelligence.

Image resource https://commons.wikimedia.org/wiki/File:Gradient_descent.gif



Let's consider a simple example of a univariate function and implement gradient descent in Python:

```
# Importing required libraries
import numpy as np
import matplotlib.pyplot as plt

# Define the objective function
def objective_function(x):
    return x**2 + 2*x + 1

# Define the gradient of the objective function
def gradient(x):
    return 2*x + 2

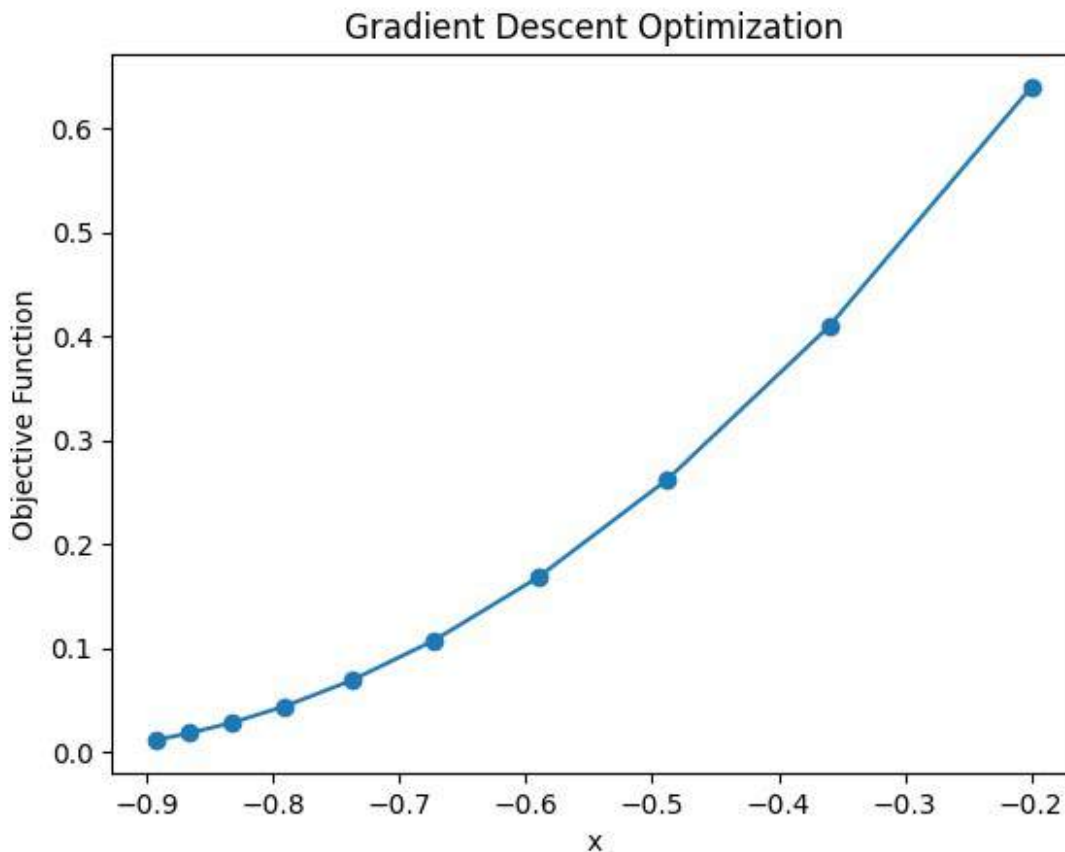
# Gradient Descent algorithm
def gradient_descent(learning_rate, num_iterations):
    x = 0.0 # Initial value of x
    x_values = []
    for i in range(num_iterations):
        gradient_value = gradient(x)
        x -= learning_rate * gradient_value
        x_values.append(x)
    return x_values

# Hyperparameters
learning_rate = 0.1
num_iterations = 10

# Perform Gradient Descent
```

```
x_values = gradient_descent(learning_rate, num_iterations)

# Plot the results
plt.plot(x_values, [objective_function(x) for x in x_values],
marker='o')
plt.xlabel('x')
plt.ylabel('Objective Function')
plt.title('Gradient Descent Optimization')
plt.show()
```



The *learning rate* is a crucial hyperparameter in gradient descent. If the learning rate is too large, the algorithm might overshoot the minimum and fail to converge. Conversely, if the learning rate is too small, convergence can be slow. Finding an appropriate learning rate is often a trial-and-error process.

2.2. Newton's Method

Newton's Method is an iterative optimization algorithm used to find the minimum (or maximum) of a function by approximating it with a quadratic model. It is a second-order optimization method that utilizes both the first-order derivative (gradient) and the second-order derivative (Hessian matrix) of the objective function. Newton's Method often converges faster than first-order methods like Gradient Descent, especially for well-behaved and smooth functions.

The *Hessian matrix* plays a crucial role in Newton's Method. It represents the curvature of the objective function and determines the step direction in the optimization process. For

Newton's Method to work effectively, the Hessian matrix should be positive-definite, meaning all its eigenvalues are positive.

Step-by-Step Explanation of Newton's Method

Step 1. Initialize: Choose an initial guess for the optimal solution, usually denoted as x_0 .

Step 2. Compute Gradient and Hessian: Calculate the gradient (first-order derivative) and the Hessian matrix (second-order derivative) of the objective function with respect to the model parameters at the current point x_k .

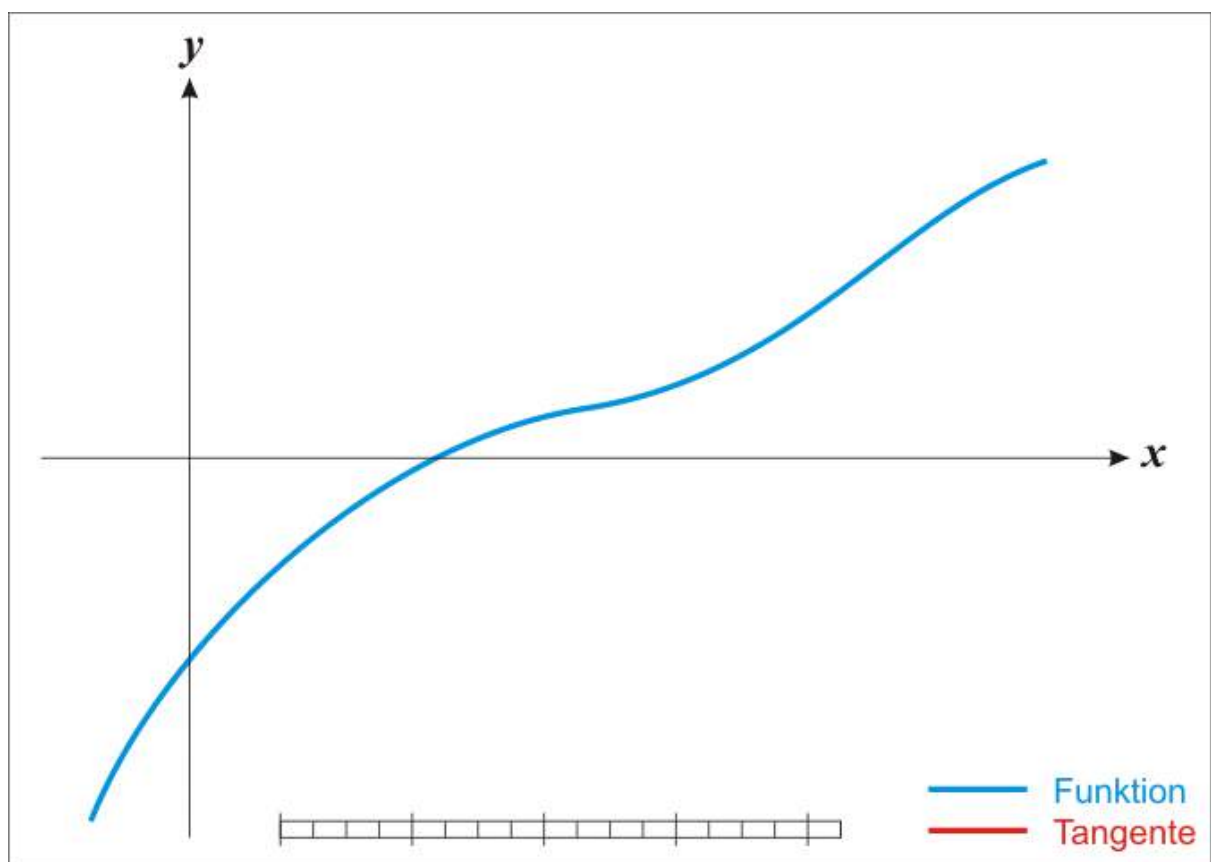
Step 3. Update Parameters: Compute the update direction, often denoted as p_k , by solving the linear equation $\text{Hessian} * p_k = -\text{gradient}$. The Hessian matrix should be positive-definite for Newton's Method to work effectively.

Step 4. Backtracking Line Search (Optional): To ensure that the objective function is decreased after each iteration, you can use line search methods to find a suitable step size, often denoted as α , in the direction p_k .

Step 5. Update Model Parameters: Update the current point using $x_{k+1} = x_k + \alpha * p_k$.

Step 6. Convergence Check: Repeat steps 2 to 5 until a convergence criterion is met. Common criteria include the change in the objective function's value being below a threshold or a fixed number of iterations.

Image resource https://commons.wikimedia.org/wiki/File:NewtonIteration_Ani.gif



Let's demonstrate Newton's Method using a simple univariate function:

```

# Importing required libraries
import numpy as np

# Define the objective function
def objective_function(x):
    return x**3 + 2*x**2 + 3 * x + 2

# Define the gradient of the objective function
def gradient(x):
    return 3 * x**2 + 4 * x + 3

# Define the Hessian matrix of the objective function
def hessian(x):
    return 6 * x + 4

# Newton's Method algorithm
def newtons_method(initial_guess, num_iterations):
    x = initial_guess
    for i in range(num_iterations):
        gradient_value = gradient(x)
        hessian_value = hessian(x)
        update_direction = -gradient_value / hessian_value
        x += update_direction
    return x

# Hyperparameters
initial_guess = 2.0
num_iterations = 5

# Perform Newton's Method
optimal_solution = newtons_method(initial_guess, num_iterations)
print("Optimal Solution:", optimal_solution)

Optimal Solution: -1.546062383341925

```

Newton's Method typically exhibits faster convergence than first-order optimization algorithms like Gradient Descent. However, it may encounter convergence issues if the Hessian matrix is not positive-definite, leading to oscillations or divergence. In such cases, modifications like the Modified Newton's Method or the Quasi-Newton methods (e.g., BFGS, L-BFGS) can be used to ensure convergence in a broader range of scenarios.

In summary, Newton's Method is a powerful optimization algorithm that can converge rapidly to the optimal solution of a smooth and well-behaved objective function when the Hessian matrix is positive-definite. Understanding its theoretical foundations and implementation details can help in effectively utilizing it for various optimization tasks.

2.3. Comparison between Gradient Descent and Newton's Method

Both Gradient Descent and Newton's Method are popular optimization algorithms used to find the minimum (or maximum) of a function. However, they differ in their approaches, convergence properties, and computational requirements. Below is a comparison between Gradient Descent and Newton's Method:

Criteria / Algorithm	Gradient Descent	Newton's Method
Approach	It is a first-order optimization algorithm that utilizes the gradient (first-order derivative) of the objective function to determine the direction of the update.	It is a second-order optimization algorithm that uses both the gradient and the Hessian matrix (second-order derivative) of the objective function to approximate it with a quadratic model and find the update direction.
Convergence Speed	It converges more slowly than Newton's Method, especially for functions with complex, curved surfaces. The convergence can be linear or superlinear.	It converges faster and more directly towards the optimal solution, often achieving quadratic convergence when the Hessian matrix is positive-definite.
Convergence Properties	It is more robust and applicable to a broader range of problems, as it does not require the Hessian matrix to be positive-definite. However, it may converge to suboptimal solutions when the objective function has narrow, elongated valleys.	It may encounter convergence issues if the Hessian matrix is not positive-definite, leading to oscillations or divergence. It requires a positive-definite Hessian for successful convergence.
Computational Complexity	It is computationally less expensive compared to Newton's Method since it only involves calculating the gradient.	It is computationally more expensive due to the calculation of both the gradient and the Hessian matrix, especially for high-dimensional problems.
Handling Large Datasets	It is more suitable for handling large datasets since it can use stochastic or mini-batch variants that update the parameters using subsets of the data.	It becomes computationally expensive for large datasets, as it requires computing the Hessian, which can be memory-intensive.
Implementation Complexity	It is straightforward to implement and requires fewer hyperparameters to tune, such as the learning rate.	It requires the computation and inversion of the Hessian matrix, which can be numerically challenging, especially for high-dimensional problems.
Robustness	It is generally more robust to noisy or ill-conditioned objective functions, as it	It can be sensitive to the starting point and may fail to converge if the initial

Criteria / Algorithm	Gradient Descent	Newton's Method
	takes smaller steps along the gradient direction.	guess is far from the optimal solution.

In summary, Gradient Descent is a simple and robust optimization algorithm that can handle a wide range of problems but may have slower convergence. On the other hand, Newton's Method provides faster convergence when the Hessian matrix is positive-definite, but it is computationally more demanding and may require careful initialization. The choice between the two methods depends on the specific characteristics of the problem and the computational resources available. Additionally, more advanced optimization algorithms, like Quasi-Newton methods (e.g., BFGS, L-BFGS), aim to combine the advantages of both Gradient Descent and Newton's Method to achieve better convergence and efficiency.

3. Constrained Optimization and Linear Programming

Constrained optimization deals with finding the optimal solution of an objective function while satisfying a set of constraints. These constraints define limitations or requirements that the solution must meet. Constrained optimization is widely used in various fields, including economics, engineering, and artificial intelligence, where complex decision-making processes require balancing multiple objectives and constraints.

Linear programming is a specific form of constrained optimization where both the objective function and constraints are linear. Linear programming problems can be formulated using mathematical notation and represented in standard form with inequalities and non-negativity constraints. The goal is to maximize or minimize the objective function while adhering to the given linear constraints.

The Simplex algorithm is a widely-used method to solve linear programming problems efficiently. Here's a step-by-step description of the Simplex algorithm:

Step 1. Convert to Standard Form: Convert the linear programming problem to standard form by introducing slack variables for inequalities and artificial variables for non-negativity constraints.

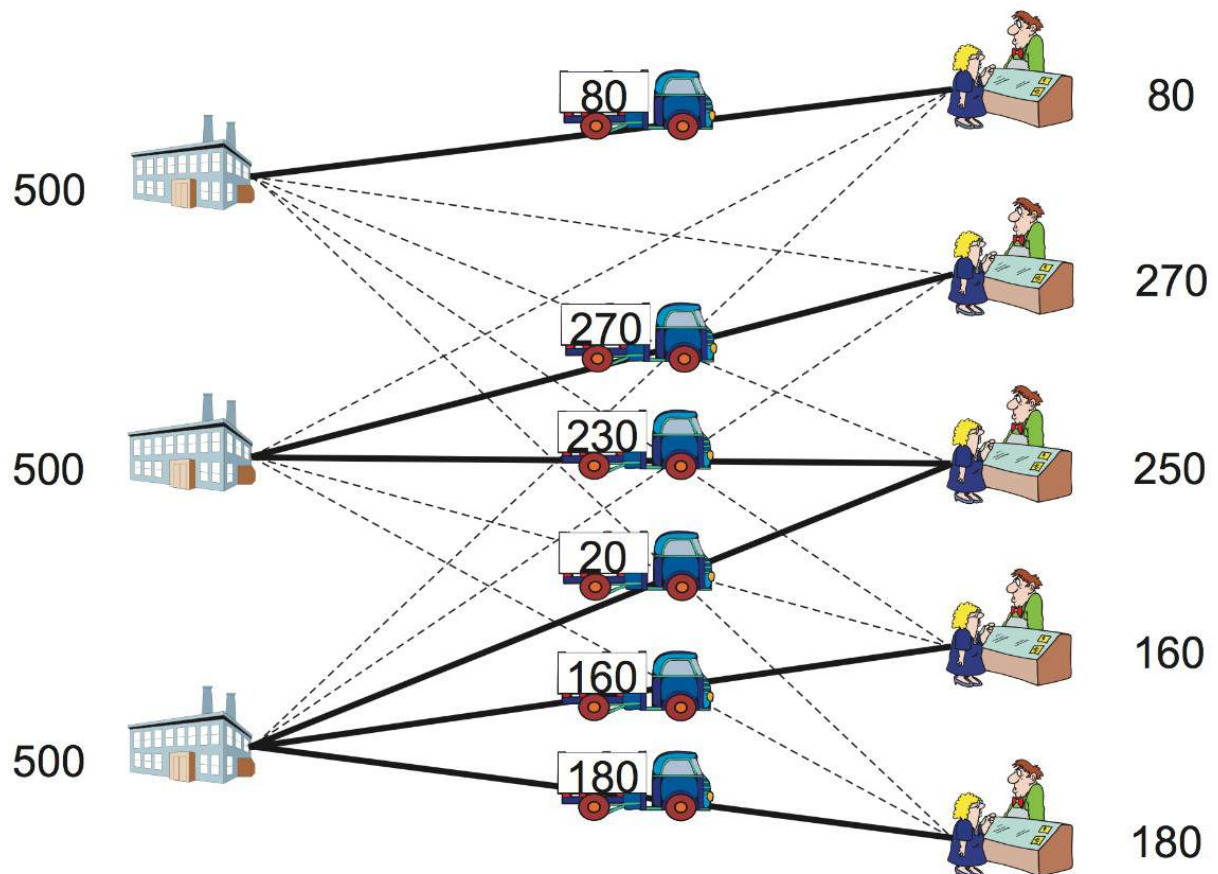
Step 2. Initialize the Simplex Tableau: Set up the initial Simplex tableau containing the objective function and constraint coefficients.

Step 3. Find Pivot Column and Row: Identify the most negative coefficient in the bottom row (objective function row) of the tableau as the pivot column. Then, select the pivot row based on the minimum non-negative ratio of the right-hand side values to the corresponding pivot column entries.

Step 4. Pivot Operation: Perform the pivot operation to make the pivot element (intersection of the pivot row and column) equal to 1 and other elements in the pivot column equal to 0.

Step 5. Iterate and Converge: Repeat steps 3 and 4 until the bottom row contains all non-negative coefficients. At convergence, the optimal solution is reached.

Image resource <https://scipbook.readthedocs.io/en/latest/intro.html>



Let's go through a step-by-step example of solving a linear programming problem and demonstrate real-world application of linear programming.

Consider the following linear programming problem:

Maximize $Z = 4x + 3y$

Subject to:

$2x + y \leq 8$

$x + 2y \leq 6$

$x, y \geq 0$

We use PuLP to formulate and solve the linear programming problem.

```
!pip install pulp

# Importing required libraries
import pulp

# Create a linear programming problem
lp_problem = pulp.LpProblem("Example_Problem", pulp.LpMaximize)

# Define variables
x = pulp.LpVariable('x', lowBound=0) # Variable x >= 0
y = pulp.LpVariable('y', lowBound=0) # Variable y >= 0
```

```

# Define objective function
lp_problem += 4*x + 3*y, "Objective"

# Define constraints
lp_problem += 2*x + y <= 8, "Constraint 1"
lp_problem += x + 2*y <= 6, "Constraint 2"

# Solve the problem
lp_problem.solve()

# Print the results
print("Optimal Solution:")
for variable in lp_problem.variables():
    print(variable.name, "=", variable.varValue)

print("Optimal Objective Value =", pulp.value(lp_problem.objective))

Collecting pulp
  Downloading PuLP-2.8.0-py3-none-any.whl (17.7 MB)
----- 17.7/17.7 MB 59.9 MB/s eta
0:00:00
al Solution:
x = 3.3333333
y = 1.3333333
Optimal Objective Value = 17.3333331

```

The output will display the optimal values of variables x and y , along with the optimal objective value (maximum value of Z).

Linear programming finds application in various real-world scenarios:

- **Resource Allocation:** Optimizing production schedules, workforce planning, or resource distribution in supply chain management.
- **Portfolio Optimization:** Constructing investment portfolios to maximize returns while adhering to risk constraints in finance.
- **Transportation and Distribution:** Finding optimal distribution of goods from suppliers to consumers while minimizing transportation costs.

Example: Suppose we want to minimize the cost of producing a certain product by optimizing the use of raw materials and labor. We can use linear programming to find the optimal allocation of resources that minimizes the cost.

```

import numpy as np
from scipy.optimize import linprog

# Define the coefficients of the objective function and the constraints
c = np.array([2, 3]) # cost of raw material and labor, respectively
A = np.array([[1, 2], [1, 2]]) # raw material and labor usage, respectively
b = np.array([5, 7]) # maximum available raw material and labor, respectively

```

```
# Define the bounds of the decision variables (i.e., the amount of
raw material and labor used)
bounds = [(1, 5), (1, 5)]

# Solve the linear programming problem to minimize the cost of
producing the product
res = linprog(c=c, A_ub=A, b_ub=b, bounds=bounds, method='highs')

# Print the optimal allocation of resources and the corresponding
minimum cost
print("Optimal allocation of resources:", res.x)
print("Minimum cost:", res.fun)

Optimal allocation of resources: [1. 1.]
Minimum cost: 5.0
```

In conclusion, constrained optimization and linear programming are powerful tools that enable us to make optimal decisions while considering various constraints. The Simplex algorithm allows us to efficiently solve linear programming problems and find optimal solutions in a wide range of real-world scenarios.

4. Combinatorial Optimization in AI

Combinatorial optimization focuses on finding optimal solutions to problems with a finite number of feasible solutions. Unlike continuous optimization, which deals with problems having infinite solution spaces, combinatorial optimization tackles discrete and often complex decision-making scenarios. The challenge lies in efficiently searching through the combinatorial space to find the best solution, making it a fundamental component in various AI applications.

Common Combinatorial Optimization Problems:

1. *Traveling Salesman Problem (TSP)*: The objective is to find the shortest possible route that visits all given cities and returns to the starting city, without visiting any city more than once.
2. *Knapsack Problem*: Given a set of items, each with a weight and value, the goal is to determine the most valuable combination of items that fits within a knapsack's weight capacity.
3. *Graph Coloring*: In this problem, the objective is to color the vertices of a graph in a way that no two adjacent vertices share the same color, using the minimum number of colors.

Let's solve the *Traveling Salesman Problem* using a brute-force approach in Python:

```
import itertools

def distance(point1, point2):
    return ((point1[0] - point2[0])**2 + (point1[1] -
point2[1])**2)**0.5
```

```

def traveling_salesman_bruteforce(points):
    min_distance = float('inf')
    optimal_path = []
    for path in itertools.permutations(points):
        total_distance = sum(distance(path[i], path[i+1]) for i in
range(len(path)-1))
        total_distance += distance(path[-1], path[0]) # Return to
the starting point
        if total_distance < min_distance:
            min_distance = total_distance
            optimal_path = path
    return optimal_path, min_distance

# Example points
points = [(0, 0), (1, 5), (4, 3), (2, 7), (8, 2), (7, 4)]

# Solve TSP using brute-force
optimal_path, min_distance = traveling_salesman_bruteforce(points)

print("Optimal Path:", optimal_path)
print("Minimum Distance:", min_distance)

Optimal Path: ((1, 5), (0, 0), (4, 3), (8, 2), (7, 4), (2, 7))
Minimum Distance: 24.525212989055323

```

Combinatorial optimization is a vital area in AI that enables us to find optimal solutions from a discrete set of possibilities. The Traveling Salesman Problem example illustrates the need for efficient algorithms to tackle complex combinatorial problems. As we continue exploring AI, understanding and applying combinatorial optimization techniques will be essential for solving real-world challenges.

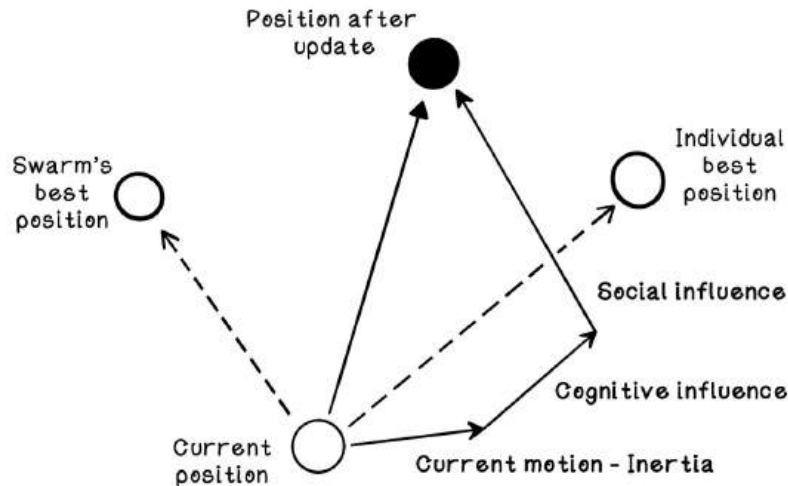
5. Evolutionary Algorithms

Evolutionary algorithms belong to the family of optimization techniques inspired by the process of natural selection. These algorithms mimic the principles of biological evolution to find optimal solutions to complex problems.

We will explore **Particle Swarm Optimization (PSO)** - a popular optimization technique inspired by the social behavior of birds and fish. PSO is a heuristic search algorithm used to find optimal solutions in complex and high-dimensional search spaces.

Particle Swarm Optimization is a population-based optimization technique introduced by Eberhart and Kennedy in 1995. It is inspired by the collective behavior of birds and fish in nature. The algorithm starts with a group of particles (potential solutions) moving through the search space. Each particle represents a potential solution and has a position and velocity. The particles explore the search space and communicate with each other to find the optimal solution.

Particle swarm optimization involves a group of individuals at different points in the solution space, all using real-life swarm concepts to find an optimal solution in the space. Imagine a swarm of bees that spread out looking for flowers and gradually converging on an area that has the most density of flowers. As more bees find the flowers, more are attracted to it. At its core, this is what particle swarm optimization entails. Particles make velocity adjustments based on an inertia component, cognitive component, and social component.



In PSO, particles form a swarm and collaborate to find the optimal solution. The swarm's behavior is governed by the following principles:

- **Position and Velocity Update:** Each particle adjusts its velocity and position based on its previous velocity, its distance to the best solution found by itself (personal best), and the distance to the best solution found by any particle in the swarm (global best).
- **Personal Best (pBest):** Each particle remembers the best position it has found so far. This is its personal best solution.
- **Global Best (gBest):** The swarm collectively maintains the best position found by any particle. This is the global best solution.

The particles use these values to adjust their velocities and positions in each iteration, allowing them to converge toward the optimal solution.

###Advantages of PSO

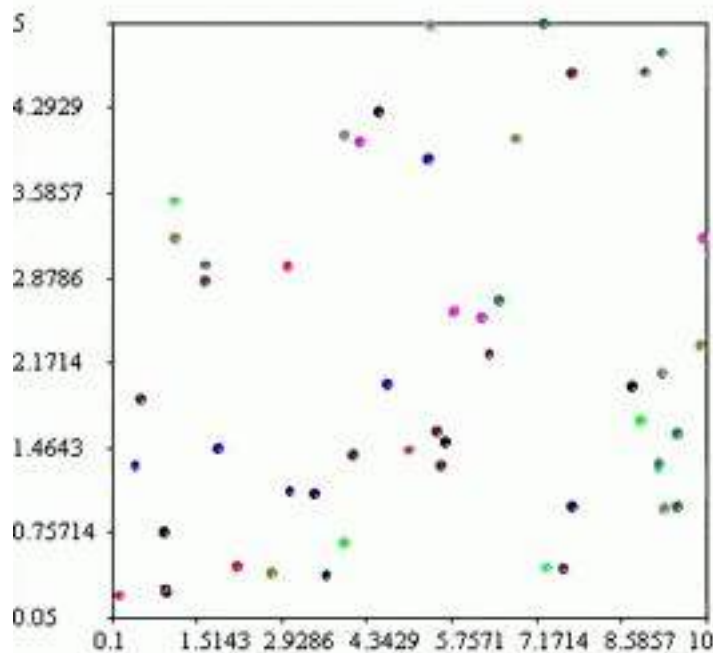
- **Ease of Implementation:** PSO is relatively simple to implement compared to other optimization algorithms, making it accessible to a wide range of users.
- **Exploration and Exploitation:** PSO effectively balances exploration and exploitation, allowing it to escape local optima while converging to the global optimum.
- **Parallelization:** The algorithm's population-based nature allows for easy parallelization, making it suitable for parallel computing environments.

Limitations of PSO

- **Convergence Issues:** PSO can struggle to converge in complex, multimodal, or discontinuous search spaces.
- **Tuning Parameters:** The performance of PSO is sensitive to the values of its parameters, and finding the right parameter values can be challenging.

Image resource

<https://commons.wikimedia.org/wiki/File:ParticleSwarmArrowsAnimation.gif>



Let's demonstrate PSO using a simple optimization problem:

```
import random

def objective_function(x):
    return x**2 - 4*x + 4

def pso(population_size, iterations):
    swarm = [random.uniform(-10, 10) for _ in
range(population_size)]
    pBest = swarm[:]
    gBest = min(swarm, key=objective_function)

    for _ in range(iterations):
        for i in range(population_size):
            velocity = random.uniform(-1, 1)
            swarm[i] += velocity

            if objective_function(swarm[i]) <
objective_function(pBest[i]):
                pBest[i] = swarm[i]

            if objective_function(swarm[i]) <
objective_function(gBest):
                gBest = swarm[i]

    return gBest

population_size = 50
iterations = 100
```

```
optimal_solution = pso(population_size, iterations)
print("Optimal Solution:", optimal_solution)
print("Optimal Objective Value:",
      objective_function(optimal_solution))
```

```
Optimal Solution: 2.0002584349643397
Optimal Objective Value: 6.678863062603568e-08
```

Let's break down the Python code example provided earlier that demonstrates the Particle Swarm Optimization (PSO) algorithm.

- `objective_function(x)` This function defines the objective function we want to optimize. In this example, we use a simple quadratic function $f(x) = x^2 - 4x + 4$. The PSO algorithm will try to find the value of x that minimizes this objective function.
- `pso(population_size, iterations)` This function implements the Particle Swarm Optimization algorithm. It takes two parameters: `population_size` and `iterations`.
- `population_size`: The number of particles (potential solutions) in the swarm.
- `iterations`: The number of iterations (generations) the PSO algorithm will run.

Here's the breakdown of the PSO algorithm within the `pso` function:

a. **Initialization:** The function initializes the swarm of particles (`swarm`) with random positions within the range $[-10, 10]$. Each particle's position represents a potential solution to the optimization problem. The personal best (`pBest`) for each particle is set to its initial position. The global best (`gBest`) is set to the position of the particle with the lowest objective function value in the initial swarm.

b. **Iteration:** The PSO algorithm runs for the specified number of `iterations`. In each iteration, each particle updates its position by adding a random velocity term (`velocity`), which represents the particle's movement in the search space.

c. **Update Personal Best (pBest):** For each particle, if its new position results in a lower objective function value than its current personal best position, the personal best is updated to the new position.

d. **Update Global Best (gBest):** Similarly, if a particle's new position results in a lower objective function value than the current global best, the global best is updated to the new position.

e. **Termination:** After all iterations, the PSO algorithm returns the position of the particle corresponding to the global best solution found during the optimization process.

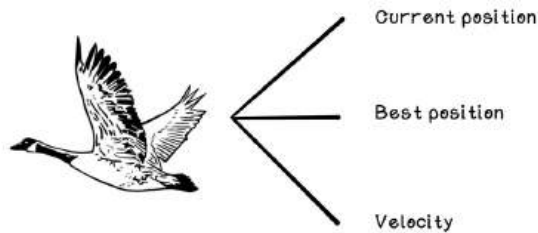
In this example, we set the `population_size` to 50 and run the PSO algorithm for 100 iterations. The optimal solution and the corresponding objective function value are printed as the output.

The optimization problem in this example is to find the value of x that minimizes the quadratic function $f(x) = x^2 - 4x + 4$. The global minimum of this function occurs at $x = 2$, where $f(2) = 0$.

After running the PSO algorithm, it should converge toward the optimal solution $x = 2$ as the global minimum of the quadratic function.

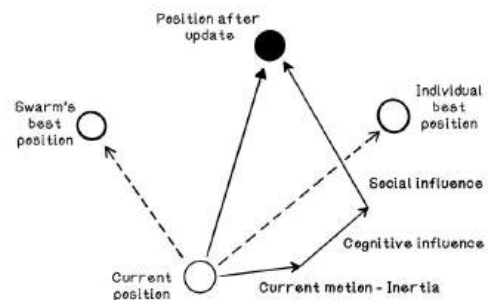
Overall, this Python code example demonstrates a simple implementation of the Particle Swarm Optimization algorithm to solve a quadratic optimization problem. PSO iteratively updates the swarm's particles to find the optimal solution that minimizes the objective function.

PSOs find good solutions in large search spaces.



Particles use their best position and the swarm's best position to move through the search space.

Adjusting the particles' velocity is the critical step of the PSO algorithm using inertia, cognitive, and social influence.



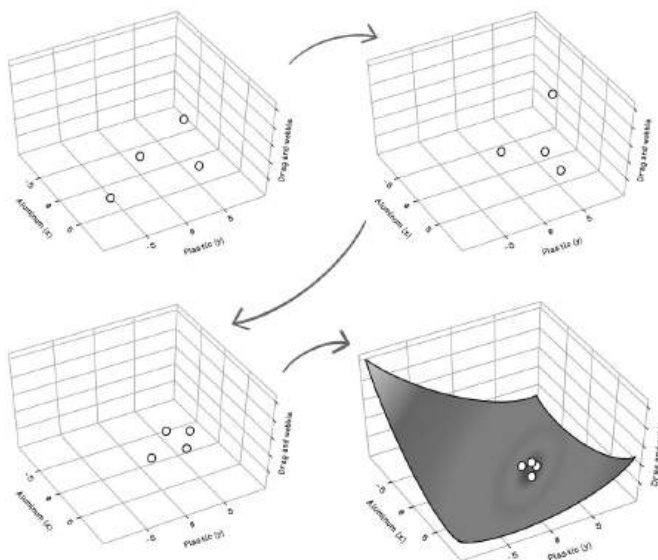
New velocity:

inertia component + social component + cognitive component

(inertia * current velocity)

(social acceleration * (swarm best position - current position))

(cognitive acceleration * (particle best position - current position))



Particles move through the search space while finding different good solutions and ideally converging on a global best solution.

Comparison between Simulated Annealing and PSO

Simulated Annealing (SA) and PSO are both heuristic optimization algorithms, but they differ in their approach:

- *Search Strategy:* SA is a single-agent method that uses a probabilistic search strategy. It accepts worse solutions with a certain probability, allowing it to explore the search space more broadly. PSO, on the other hand, is a population-based method that explores the search space through a swarm of particles collaborating to find the optimal solution.
- *Exploration vs. Exploitation:* SA tends to emphasize exploration, making it suitable for escaping local optima, while PSO effectively balances exploration and exploitation, striking a better trade-off between exploration and exploitation.
- *Parameter Sensitivity:* Both algorithms have parameters to tune. SA's temperature schedule and PSO's parameters, such as inertia weight and acceleration coefficients, can significantly impact their performance.

In general, SA is useful for problems with rugged or discontinuous search spaces, while PSO is well-suited for smooth and continuous search spaces.

Particle Swarm Optimization is a powerful optimization technique that simulates the social behavior of particles to find optimal solutions. With the Python code example and comparison to Simulated Annealing, you now have a good understanding of how PSO works and its advantages in optimization tasks.

6. Real-world Applications of Optimization in AI

Optimization techniques play a crucial role in various real-world applications of Artificial Intelligence. Let's explore some key areas where optimization is extensively used in AI:

1. Resource Allocation and Scheduling

AI-powered optimization is widely employed in industries where efficient resource allocation and scheduling are critical. Some applications include:

- *Supply Chain Management:* Optimizing inventory levels, production schedules, and distribution routes to minimize costs and improve efficiency.
- *Transportation and Logistics:* Optimizing vehicle routing, delivery schedules, and load distribution for faster and cost-effective transportation.
- *Healthcare:* Scheduling surgeries, appointments, and staff shifts to maximize resource utilization and minimize waiting times.

2. Portfolio Optimization and Financial Modeling

In finance, optimization is utilized for portfolio management, risk assessment, and financial modeling:

- *Portfolio Optimization:* Optimizing asset allocation to maximize returns while minimizing risk, considering factors like asset correlation and constraints.

- *Option Pricing Models:* Using optimization techniques to calibrate option pricing models like the Black-Scholes model to market data.
- *Risk Management:* Employing optimization to model and manage risk exposure in financial portfolios.

3. Game Playing and Strategy Development

AI optimization techniques are employed to create sophisticated game-playing agents and develop winning strategies:

- *Chess and Go:* AI-powered engines use optimization algorithms to search for the best moves and anticipate opponent strategies.
- *Real-Time Strategy (RTS) Games:* AI agents use optimization to plan unit movements, resource management, and combat strategies.
- *Sports Analytics:* Optimization is used to develop optimal game plans, lineups, and player substitutions in sports like soccer, basketball, and baseball.

4. Neural Network Training and Hyperparameter Optimization

In the realm of machine learning and deep learning, optimization is vital for neural network training:

- *Gradient Descent:* Optimization algorithms like Stochastic Gradient Descent (SGD) are used to minimize the loss function during neural network training.
- *Hyperparameter Optimization:* Techniques like Bayesian optimization and random search are employed to tune hyperparameters for better model performance.
- *Model Architecture Search:* Using optimization to find optimal neural network architectures, such as in Neural Architecture Search (NAS).

5. Routing and Network Optimization

In networking and telecommunications, optimization is applied to improve efficiency and performance:

- *Network Routing:* Optimizing data packet routing to minimize latency, maximize bandwidth, and avoid network congestion.
- *Wireless Sensor Networks:* Using optimization to determine optimal sensor placement and coverage for efficient data collection.
- *Telecommunication Planning:* Optimizing the placement of towers and antennas to maximize coverage and minimize signal interference.

Optimization in AI has a vast array of real-world applications, ranging from resource allocation and scheduling to portfolio optimization, game playing, and deep learning. These techniques enable AI systems to make informed decisions, solve complex problems, and achieve optimal outcomes in diverse domains.

As AI continues to advance, optimization will remain a critical component, driving innovations and efficiency across industries and applications.

7. Q&A and Wrap-up

The session on "Optimization in AI" covered key concepts essential for understanding optimization techniques and their applications in the field of Artificial Intelligence. Optimization is the process of finding the best solution from a set of possible solutions to a specific problem. It plays a vital role in various AI applications, such as resource allocation, scheduling, portfolio optimization, and game playing. Techniques like Gradient Descent, Newton's Method, Evolutionary Algorithms, and Particle Swarm Optimization were introduced, each offering unique ways to tackle different optimization problems. The session also highlighted real-world applications of optimization in AI, showcasing its significance in improving efficiency, decision-making, and problem-solving across diverse domains.

#Optimization. Practical Tasks

Please, execute two rows of code below at first.

```
!pip install git+https://github.com/mehalya/cooltest.git
from cooltest.test_cool_3 import *
```

Task 1. Resource Scheduling

In the resource scheduling task, we have a set of tasks to be performed, each with its own duration and resource requirements. Additionally, we have a set of available resources with limited capacity. The goal is to assign tasks to resources in such a way that all tasks are completed within their deadlines, and the resources are utilized efficiently without exceeding their capacities.

Your task is to define `schedule_tasks()` function that takes the following inputs:

- **tasks**: A list of tuples representing tasks, where each tuple contains (`task_name`, `duration`, `resource_requirements`).
- **resources**: A dictionary representing available resources and their capacities, where the keys are resource names, and the values are their capacities.
- **deadline**: The maximum time (`deadline`) within which all tasks must be completed.

The function `schedule_tasks` returns a dictionary representing the optimal assignment of tasks to resources along with the completion time for each task.

Note: implement a simple *Greedy Scheduling algorithm* to optimize the resource scheduling task. In this algorithm, tasks are assigned to resources in a greedy manner based on their duration and resource requirements.

```
@test_schedule_task
def schedule_tasks(tasks, resources, deadline):
    """
    Optimize resource scheduling to complete tasks within the given
    deadline using Greedy Scheduling.

    Args:
        tasks (list of tuple): A list of tasks, where each tuple
        contains (task_name, duration, resource_requirements).
        resources (dict): A dictionary representing available
        resources and their capacities.
        Keys are resource names, and values are
        their capacities.
        deadline (float): The maximum time (deadline) within which
        all tasks must be completed.

    Returns:
        dict: A dictionary representing the optimal assignment of
        tasks to resources.
        The keys are resource names, and the values are lists
```

```

of tasks assigned to each resource.
    The dictionary also includes the completion time for
each task.
    Example: {'Resource1': ['TaskA', 'TaskB'],
'Resource2': ['TaskC'], 'TaskA': 4.5, 'TaskB': 7.2, 'TaskC': 5.0}
    """
    #initialization
    assigned_tasks = {}
    task_times = {}
    current_time = 0.0

    # Sort tasks in ascending order of their durations

    # Sort resources in ascending order of their remaining
capacities

    # Assign task to the resource

    # Update resource capacities

    # If the task couldn't be assigned to any resource,
extend the deadline

    # Update completion time for each task

# Example usage:
tasks_list = [
    ('TaskA', 4.5, {'Resource1': 2, 'Resource2': 1}),
    ('TaskB', 7.2, {'Resource2': 3}),
    ('TaskC', 5.0, {'Resource1': 1})
]

resources_dict = {'Resource1': 10, 'Resource2': 15}
deadline_time = 12.0

result = schedule_tasks(tasks_list, resources_dict, deadline_time)
print(result)

```

Task 2. Vehicle Routing Problem (VRP)

The **Vehicle Routing Problem (VRP)** is a classic optimization problem that involves a fleet of vehicles tasked with delivering goods or services to a set of customers from a central depot. Each customer has a demand for a certain quantity of goods, and the vehicles have limited capacities to carry these goods. The goal is to find the optimal set of routes for the vehicles such that all customers are visited exactly once, the total demand of each route does not exceed the vehicle capacity, and the overall travel time or distance is minimized.

Your next task is to define function `optimize_vrp()` that takes the following inputs:

- **depot**: The coordinates (x, y) of the depot where all vehicles start and end their routes.
- **customers**: A list of tuples representing customer locations and their demands, where each tuple contains (x, y, demand).
- **vehicle_capacity**: The maximum capacity of each vehicle.
- **num_vehicles**: The number of vehicles available in the fleet.

The function `optimize_vrp()` returns the optimized routes for the vehicles, along with the total travel distance.

Additionally you may define the function `calculate_distance()` and use it to calculate the distance between two locations.

Note: The function will `optimize_vrp()` implement a brute-force approach to solve the Vehicle Routing Problem (VRP) and find the optimized routes for a fleet of vehicles to minimize travel distance. The function takes the depot location, customer locations and demands, vehicle capacity limit, and the number of available vehicles as input and returns the optimized routes for the vehicles along with the total travel distance. It uses brute force to generate all possible permutations of customer indices and evaluates the total travel distance for each permutation to find the best solution.

```
import itertools
import math

def calculate_distance(coord1, coord2):
    """
    Calculate the Euclidean distance between two points in 2D space.

    Args:
        coord1 (tuple): The coordinates (x, y) of the first point.
        coord2 (tuple): The coordinates (x, y) of the second point.

    Returns:
        float: The Euclidean distance between the two points.
    """

    return result

@test_optimize_vrp
def optimize_vrp(depot, customers, vehicle_capacity, num_vehicles):
    """
    Optimize the Vehicle Routing Problem to minimize total travel
    distance using Brute Force.

    Args:
        depot (tuple): The coordinates (x, y) of the depot, where
        the vehicles start and end their routes.
        customers (list of tuple): A list of tuples representing the
        coordinates (x, y) of each customer location.
        vehicle_capacity (int): The maximum capacity of each
        vehicle.
        num_vehicles (int): The number of vehicles available in the
```

```

fleet.

    Returns:
        list: A list of routes, where each route represents the
              sequence of customer locations visited by a single vehicle.
        """

    # Generate all possible permutations of customer visits

    # Calculate distance from last customer in the
    route to the current customer

    # Find the route with the minimum total distance

    return result

# Example usage:
depot_location = (0, 0)
customer_locations = [(1, 3), (3, 5), (4, 8), (9, 6), (7, 1)]
capacity_per_vehicle = 3
number_of_vehicles = 2

optimized_routes = optimize_vrp(depot_location, customer_locations,
                                capacity_per_vehicle, number_of_vehicles)
print(optimized_routes)

```

Task 3. Inventory Management

Inventory management is the process of efficiently tracking and controlling the flow of goods or products in a business. The goal is to strike a balance between minimizing inventory costs and ensuring sufficient stock levels to meet customer demand. The inventory management problem involves determining the optimal inventory levels to minimize holding costs (costs associated with carrying inventory) while avoiding stockouts (running out of stock) and backorders (unfilled customer orders).

Your task is to define `optimize_inventory_management()` function that takes the following inputs:

- **demand**: A list representing the demand for each period (e.g., month, week) in the planning horizon.
- **holding_cost**: The cost of holding one unit of inventory for one period (e.g., month, week).
- **ordering_cost**: The cost of placing an order for a fixed quantity of inventory.
- **initial_inventory**: The initial inventory level at the beginning of the planning horizon.
- **reorder_point**: The inventory level at which a new order should be placed to avoid stockouts.

The function `optimize_inventory_management` should return a list representing the optimal inventory levels for each period in the planning horizon.

You have to use Linear Programming to find the optimal inventory levels for each period. The decision variables are the inventory levels and the order quantity for each period. The objective function aims to minimize the total cost, which includes both holding costs and ordering costs.

Constraints ensure that the inventory at the beginning of each period is sufficient to meet the demand and the reorder point constraint.

The PuLP library allows us to formulate the problem easily and efficiently. Once the Linear Programming problem is defined, we call `model.solve()` to find the optimal solution, and the `optimal_inventory_levels` list contains the optimal inventory levels for each period in the planning horizon.

Linear Programming Model: Decision Variables:

- `inventory[period]`: The inventory level at the beginning of each period.
- `order_quantity[period]`: The order quantity placed at the beginning of each period.

Objective Function:

- Minimize the total cost, which includes holding costs and ordering costs for each period.

Constraints:

- `inventory[0] == initial_inventory`: Initial inventory level constraint.
- `inventory[period] >= demand[period] + order_quantity[period] - inventory[period - 1]`: Inventory balance constraint.
- `inventory[period] >= reorder_point`: Reorder point constraint.
- `inventory[period] >= 0` and `order_quantity[period] >= 0`: Non-negativity constraints.

Note:

- The `demand` list should contain the demand for each period in the planning horizon.
- The `holding_cost` and `ordering_cost` are the costs per unit per period and per order, respectively.
- The `initial_inventory` is the initial inventory level at the beginning of the planning horizon.
- The `reorder_point` is the inventory level at which a new order should be placed.
- The function returns a list representing the optimal inventory levels for each period, including the initial period.

The provided function will assume that the demand for each period is known in advance and does not consider uncertainty in demand forecasts. Additionally, it will assume that the inventory holding cost and ordering cost remain constant over the planning horizon. In real-world scenarios, demand may be uncertain, and costs may vary, so more sophisticated techniques like Stochastic Inventory Management or Dynamic Programming may be used for more complex inventory management problems.

```
!pip install pulp
```

```

import pulp

@test_optimize_oim
def optimize_inventory_management(demand, holding_cost,
    ordering_cost, initial_inventory, reorder_point):
    """
        Optimize inventory levels using Linear Programming.

        This function uses Linear Programming to determine the optimal
        inventory levels for each period in the planning
        horizon. The goal is to minimize the total cost of inventory
        while meeting the demand and inventory requirements.

        Args:
            demand (list): A list representing the demand for each
            period.
            holding_cost (float): The cost of holding one unit of
            inventory for one period.
            ordering_cost (float): The cost of placing an order for a
            fixed quantity of inventory.
            initial_inventory (int): The initial inventory level at the
            beginning of the planning horizon.
            reorder_point (int): The inventory level at which a new
            order should be placed to avoid stockouts.

        Returns:
            list: A list representing the optimal inventory levels for
            each period.
    """

    # Create a Linear Programming problem

    # Decision variables

    # Objective function: minimize total cost

    # Constraints

    # Reorder point constraint

    # Solve the Linear Programming problem

    # Extract the optimal solution

    return result

# Example usage:
demand_forecast = [10, 20, 15, 25, 30]
holding_cost_per_period = 1.5

```

```
ordering_cost_per_order = 25.0
initial_inventory_level = 50
reorder_point_level = 50

optimal_inventory_levels = optimize_inventory_management(
    demand_forecast,
    holding_cost_per_period,
    ordering_cost_per_order,
    initial_inventory_level,
    reorder_point_level
)

print("Optimal Inventory Levels:", optimal_inventory_levels)
```

Topic: Machine Learning in AI

Duration: 6 hours

Agenda:

1. Introduction to Machine Learning
 - a. Understanding the basics of machine learning
 - b. Differentiating between supervised, unsupervised, and reinforcement learning
 - c. Real-world applications of machine learning in AI
2. Basics of Probability
 - a. Probability fundamentals
 - b. Conditional probability and Bayes' theorem
 - c. Probability distributions (e.g., Gaussian, Bernoulli)
 - d. Incorporating probability in machine learning models
3. Supervised Learning
 - a. Supervised learning concept and workflow
 - b. Linear Regression
 - Understanding linear regression
 - Cost function and gradient descent
 - Evaluation metrics (e.g., mean squared error)
 - c. Classification Algorithms
 - Logistic Regression
 - k-Nearest Neighbors (k-NN)
 - Decision Trees
 - d. Ensemble Methods
 - Random Forest
 - Boosting techniques (e.g., AdaBoost)
 - e. Hands-on: Building a supervised learning model in Python
4. Unsupervised Learning
 - a. Unsupervised learning concept and workflow
 - b. Clustering Algorithms
 - k-Means Clustering
 - Hierarchical Clustering

- c. Dimensionality Reduction
 - Principal Component Analysis (PCA)
 - t-distributed Stochastic Neighbor Embedding (t-SNE)
 - d. Hands-on: Applying unsupervised learning algorithms to real datasets
 - 5. Hypothesis Testing and Experimental Design
 - a. Introduction to hypothesis testing
 - b. The p-value and significance level
 - c. Experimental design principles in machine learning
 - d. A/B testing and its applications
 - 6. Recommender Systems
 - a. Understanding recommender systems
 - b. Collaborative filtering and content-based filtering approaches
 - c. Evaluation metrics for recommender systems
 - 7. Q&A and Wrap-up
 - a. Answering participant questions and clarifications
 - b. Recapitulation of the essential concepts covered during the course
 - c. Encouraging further learning and resources for exploring advanced machine learning topics
-

Introduction to Machine Learning

- a. Understanding the basics of machine learning

An introduction to Machine Learning

Arthur Samuel, an early American leader in the field of computer gaming and artificial intelligence, coined the term “Machine Learning” in 1959 while at IBM. He defined machine learning as “the field of study that gives computers the ability to learn without being explicitly programmed”. However, there is no universally accepted definition for machine learning. Different authors define the term differently. We give below two more definitions.

- Machine learning is programming computers to optimize a performance criterion using example data or past experience. We have a model defined up to some parameters, and learning is the execution of a computer program to optimize the parameters of the model using the training data or past experience. The model may be predictive to make predictions in the future, or descriptive to gain knowledge from data.

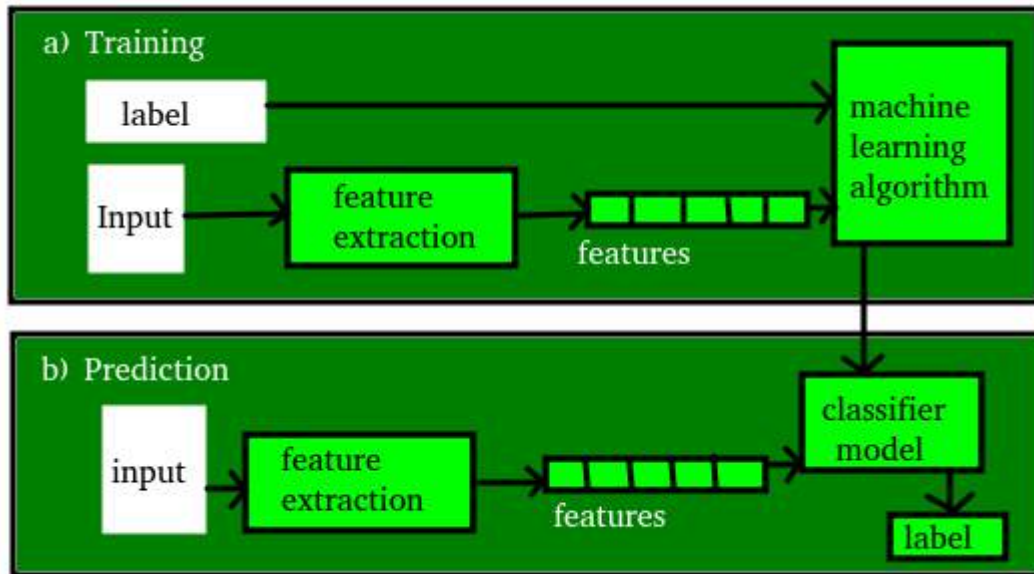
- The field of study known as machine learning is concerned with the question of how to construct computer programs that automatically improve with experience.

Machine learning is a subfield of artificial intelligence that involves the development of algorithms and statistical models that enable computers to improve their performance in tasks through experience. These algorithms and models are designed to learn from data and make predictions or decisions without explicit instructions. There are several types of machine learning, including supervised learning, unsupervised learning, and reinforcement learning. Supervised learning involves training a model on labeled data, while unsupervised learning involves training a model on unlabeled data. Reinforcement learning involves training a model through trial and error. Machine learning is used in a wide variety of applications, including image and speech recognition, natural language processing, and recommender systems.

Terminologies of Machine Learning

- **Model** A model is a specific representation learned from data by applying some machine learning algorithm. A model is also called a hypothesis.
- **Feature** A feature is an individual measurable property of our data. A set of numeric features can be conveniently described by a feature vector. Feature vectors are fed as input to the model. For example, in order to predict a fruit, there may be features like color, smell, taste, etc. Note: Choosing informative, discriminating and independent features is a crucial step for effective algorithms. We generally employ a feature extractor to extract the relevant features from the raw data.
- **Target (Label)** A target variable or label is the value to be predicted by our model. For the fruit example discussed in the features section, the label with each set of input would be the name of the fruit like apple, orange, banana, etc.
- **Training** The idea is to give a set of inputs(features) and its expected outputs(labels), so after training, we will have a model (hypothesis) that will then map new data to one of the categories trained on.
- **Prediction** Once our model is ready, it can be fed a set of inputs to which it will provide a predicted output(label). But make sure if the machine performs well on unseen data, then only we can say the machine performs well.

The figure shown below clears the above concepts:



Here are the steps to get started with machine learning:

1. **Define the Problem:** Identify the problem you want to solve and determine if machine learning can be used to solve it.
2. **Collect Data:** Gather and clean the data that you will use to train your model. The quality of your model will depend on the quality of your data.
3. **Explore the Data:** Use data visualization and statistical methods to understand the structure and relationships within your data.
4. **Pre-process the Data:** Prepare the data for modeling by normalizing, transforming, and cleaning it as necessary.
5. **Split the Data:** Divide the data into training and test datasets to validate your model.
6. **Choose a Model:** Select a machine learning model that is appropriate for your problem and the data you have collected.
7. **Train the Model:** Use the training data to train the model, adjusting its parameters to fit the data as accurately as possible.
8. **Evaluate the Model:** Use the test data to evaluate the performance of the model and determine its accuracy.
9. **Fine-tune the Model:** Based on the results of the evaluation, fine-tune the model by adjusting its parameters and repeating the training process until the desired level of accuracy is achieved.
10. **Deploy the Model:** Integrate the model into your application or system, making it available for use by others.

11. Monitor the Model: Continuously monitor the performance of the model to ensure that it continues to provide accurate results over time.

Example :

Here is a simple machine-learning example in Python that demonstrates how to train a model to predict the species of iris flowers based on their sepal and petal measurements:

```
# load the iris dataset
from sklearn.datasets import load_iris
iris = load_iris()

# store the feature matrix (X) and response vector (y)
X = iris.data
y = iris.target

# splitting X and y into training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.4, random_state=1)

# training the model on training set
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
gnb.fit(X_train, y_train)

# making predictions on the testing set
y_pred = gnb.predict(X_test)

# comparing actual response values (y_test) with predicted response
values (y_pred)
from sklearn import metrics
print("Gaussian Naive Bayes model accuracy(in %):",
metrics.accuracy_score(y_test, y_pred)*100)

Gaussian Naive Bayes model accuracy(in %): 95.0
```

b. Differentiating between supervised, unsupervised, and reinforcement learning

Definition of learning:

A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks T, as measured by P, improves with experience E.

Examples

- Handwriting recognition learning problem
- - Task T : Recognizing and classifying handwritten words within images
- - Performance P : Percent of words correctly classified

- - Training experience E : A dataset of handwritten words with given classifications
- A robot driving learning problem
- - Task T : Driving on highways using vision sensors
- - Performance P : Average distance traveled before an error
- - Training experience E : A sequence of images and steering commands recorded while observing a human driver

Definition: A computer program which learns from experience is called a machine learning program or simply a learning program .

Classification of Machine Learning

Machine learning implementations are classified into four major categories, depending on the nature of the learning “signal” or “response” available to a learning system which are as follows:

A. Supervised learning:

Supervised learning is the machine learning task of learning a function that maps an input to an output based on example input-output pairs. The given data is labeled. Both classification and regression problems are supervised learning problems.

- Example — Consider the following data regarding patients entering a clinic . The data consists of the gender and age of the patients and each patient is labeled as “healthy” or “sick”.

Gender	Age	Label
M	48	sick
M	67	sick
F	53	healthy
M	49	sick
F	32	healthy
M	34	healthy
M	21	healthy

B. Unsupervised learning:

Unsupervised learning is a type of machine learning algorithm used to draw inferences from datasets consisting of input data without labeled responses. In unsupervised learning algorithms, classification or categorization is not included in the observations. Example: Consider the following data regarding patients entering a clinic. The data consists of the gender and age of the patients.

Gender	Age
M	48
M	67

Gender	Age
F	53
M	49
F	32
M	34
M	21

As a kind of learning, it resembles the methods humans use to figure out that certain objects or events are from the same class, such as by observing the degree of similarity between objects. Some recommendation systems that you find on the web in the form of marketing automation are based on this type of learning.

C. Reinforcement learning:

Reinforcement learning is the problem of getting an agent to act in the world so as to maximize its rewards.

A learner is not told what actions to take as in most forms of machine learning but instead must discover which actions yield the most reward by trying them. For example — Consider teaching a dog a new trick: we cannot tell him what to do, what not to do, but we can reward/punish it if it does the right/wrong thing.

D. Semi-supervised learning:

Where an incomplete training signal is given: a training set with some (often many) of the target outputs missing. There is a special case of this principle known as Transduction where the entire set of problem instances is known at learning time, except that part of the targets are missing. Semi-supervised learning is an approach to machine learning that combines small labeled data with a large amount of unlabeled data during training. Semi-supervised learning falls between unsupervised learning and supervised learning.

Categorizing based on Required Output

Another categorization of machine-learning tasks arises when one considers the desired output of a machine-learned system:

- **Classification:** When inputs are divided into two or more classes, the learner must produce a model that assigns unseen inputs to one or more (multi-label classification) of these classes. This is typically tackled in a supervised way. Spam filtering is an example of classification, where the inputs are email (or other) messages and the classes are “spam” and “not spam”.
- **Regression:** Which is also a supervised problem, A case when the outputs are continuous rather than discrete.
- **Clustering:** When a set of inputs is to be divided into groups. Unlike in classification, the groups are not known beforehand, making this typically an unsupervised task.

Machine Learning comes into the picture when problems cannot be solved using typical approaches. ML algorithms combined with new computing technologies promote scalability and improve efficiency. Modern ML models can be used to make predictions ranging from outbreaks of disease to the rise and fall of stocks.

Applications of Machine Learning

Machine learning is one of the most exciting technologies that one would have ever come across. As is evident from the name, it gives the computer that which makes it more similar to humans: The ability to learn. Machine learning is actively being used today, perhaps in many more places than one would expect.

Today, companies are using Machine Learning to improve business decisions, increase productivity, detect disease, forecast weather, and do many more things. With the exponential growth of technology, we not only need better tools to understand the data we currently have, but we also need to prepare ourselves for the data we will have. To achieve this goal we need to build intelligent machines. We can write a program to do simple things. But most of the time, Hardwiring Intelligence in it is difficult. The best way to do it is to have some way for machines to learn things themselves. A mechanism for learning – if a machine can learn from input then it does the hard work for us. This is where Machine Learning comes into action. Some of the most common examples are:

- Image Recognition
- Speech Recognition
- Recommender Systems
- Fraud Detection
- Self Driving Cars
- Medical Diagnosis
- Stock Market Trading
- Virtual Try On

Image Recognition

Image Recognition is one of the reasons behind the boom one could have experienced in the field of Deep Learning. The task which started from classification between cats and dog images has now evolved up to the level of Face Recognition and real-world use cases based on that like employee attendance tracking.

Also, image recognition has helped revolutionized the healthcare industry by employing smart systems in disease recognition and diagnosis methodologies.

Speech Recognition

Speech Recognition based smart systems like Alexa and Siri have certainly come across and used to communicate with them. In the backend, these systems are based basically on Speech Recognition systems. These systems are designed such that they can convert voice instructions into text.

One more application of the Speech recognition that we can encounter in our day-to-day life is that of performing Google searches just by speaking to it.

Recommender Systems

As our world has digitalized more and more approximately every tech giants try to provide customized services to its users. This application is possible just because of the recommender systems which can analyze a user's preferences and search history and based on that they can recommend content or services to them.

An example of these services is very common for example youtube. It recommends new videos and content based on the user's past search patterns. Netflix recommends movies and series based on the interest provided by users when someone creates an account for the very first time.

Fraud Detection

In today's world, most things have been digitalized varying from buying toothbrushes or making transactions of millions of dollars everything is accessible and easy to use. But with this process of digitization cases of fraudulent transactions and fraudulent activities have increased. Identifying them is not that easy but machine learning systems are very efficient in these tasks.

Due to these applications only whenever the system detects red flags in a user's activity than a suitable notification be provided to the administrator so, that these cases can be monitored properly for any spam or fraud activities.

Self Driving Cars

It would have been assumed that there is certainly some ghost who is driving a car if we ever saw a car being driven without a driver but all thanks to machine learning and deep learning that in today's world, this is possible and not a story from some fictional book. Even though the algorithms and tech stack behind these technologies are highly advanced but at the core it is machine learning which has made these applications possible.

The most common example of this use case is that of the Tesla cars which are well-tested and proven for autonomous driving.

Medical Diagnosis

If you are a machine learning practitioner or even if you are a student then you must have heard about projects like breast cancer Classification, Parkinson's Disease Classification, Pneumonia detection, and many more health-related tasks which are performed by machine learning models with more than 90% of accuracy.

Not even in the field of disease diagnosis in human beings but they work perfectly fine for plant disease-related tasks whether it is to predict the type of disease it is or to detect whether some disease is going to occur in the future.

Stock Market Trading

Stock Market has remained a hot topic among working professionals and even students because if you have sufficient knowledge of the markets and the forces which drives them then you can make fortune in this domain. Attempts have been made to create intelligent systems which can predict future price trends and market value as well.

This can be considered as one of the applications of time series forecasting because stock price data is nothing but sequential data in which the time at which data has been taken is of utmost importance.

Virtual Try On

Have you ever purchased your specs or lenses from Lenskart? If yes then you must have come across its feature where you can try different frames virtually without actually purchasing them or visiting the outlet. This has become possible just because of the machine learning systems only which identify certain landmarks on a person's face and then place the specs virtually on your face using those landmarks.

Basics of Probability (Duration: 30 minutes)

a. Probability fundamentals

b. Conditional probability and Bayes' theorem

c. Probability distributions (e.g., Gaussian, Bernoulli)

d. Incorporating probability in machine learning models

Probabilistic Models in Machine Learning

Machine learning algorithms today rely heavily on probabilistic models, which take into consideration the uncertainty inherent in real-world data. These models make predictions based on probability distributions, rather than absolute values, allowing for a more nuanced and accurate understanding of complex systems. One common approach is Bayesian inference, where prior knowledge is combined with observed data to make predictions. Another approach is maximum likelihood estimation, which seeks to find the model that best fits observational data.

What are Probabilistic Models? Probabilistic models are an essential component of machine learning, which aims to learn patterns from data and make predictions on new, unseen data. They are statistical models that capture the inherent uncertainty in data and incorporate it into their predictions. Probabilistic models are used in various applications such as image and speech recognition, natural language processing, and recommendation systems. In recent years, significant progress has been made in developing probabilistic models that can handle large datasets efficiently.

Categories Of Probabilistic Models These models can be classified into the following categories:

- Generative models
- Discriminative models.
- Graphical models

Generative models: Generative models aim to model the joint distribution of the input and output variables. These models generate new data based on the probability distribution of the original dataset. Generative models are powerful because they can generate new data that resembles the training data. They can be used for tasks such as image and speech synthesis, language translation, and text generation.

Discriminative models The discriminative model aims to model the conditional distribution of the output variable given the input variable. They learn a decision boundary that separates the different classes of the output variable. Discriminative models are useful when the focus is on making accurate predictions rather than generating new data. They can be used for tasks such as image recognition, speech recognition, and sentiment analysis.

Graphical models These models use graphical representations to show the conditional dependence between variables. They are commonly used for tasks such as image recognition, natural language processing, and causal inference.

Naive Bayes Algorithm in Probabilistic Models The Naive Bayes algorithm is a widely used approach in probabilistic models, demonstrating remarkable efficiency and effectiveness in solving classification problems. By leveraging the power of the Bayes theorem and making

simplifying assumptions about feature independence, the algorithm calculates the probability of the target class given the feature set. This method has found diverse applications across various industries, ranging from spam filtering to medical diagnosis. Despite its simplicity, the Naive Bayes algorithm has proven to be highly robust, providing rapid results in a multitude of real-world problems.

Naive Bayes is a probabilistic algorithm that is used for classification problems. It is based on the Bayes theorem of probability and assumes that the features are conditionally independent of each other given the class. The Naive Bayes Algorithm is used to calculate the probability of a given sample belonging to a particular class. This is done by calculating the posterior probability of each class given the sample and then selecting the class with the highest posterior probability as the predicted class.

Probabilistic Models in Deep Learning Deep learning, a subset of machine learning, also relies on probabilistic models. Probabilistic models are used to optimize complex models with many parameters, such as neural networks. By incorporating uncertainty into the model training process, deep learning algorithms can provide higher accuracy and generalization capabilities. One popular technique is variational inference, which allows for efficient estimation of posterior distributions.

Importance of Probabilistic Models

- Probabilistic models play a crucial role in the field of machine learning, providing a framework for understanding the underlying patterns and complexities in massive datasets.
- Probabilistic models provide a natural way to reason about the likelihood of different outcomes and can help us understand the underlying structure of the data.
- Probabilistic models help enable researchers and practitioners to make informed decisions when faced with uncertainty.
- Probabilistic models allow us to perform Bayesian inference, which is a powerful method for updating our beliefs about a hypothesis based on new data. This can be particularly useful in situations where we need to make decisions under uncertainty.

Advantages Of Probabilistic Models

- Probabilistic models are an increasingly popular method in many fields, including artificial intelligence, finance, and healthcare.
- The main advantage of these models is their ability to take into account uncertainty and variability in data. This allows for more accurate predictions and decision-making, particularly in complex and unpredictable situations.
- Probabilistic models can also provide insights into how different factors influence outcomes and can help identify patterns and relationships within data.

Disadvantages Of Probabilistic Models

There are also some disadvantages to using probabilistic models.

- One of the disadvantages is the potential for overfitting, where the model is too specific to the training data and doesn't perform well on new data.
- Not all data fits well into a probabilistic framework, which can limit the usefulness of these models in certain applications.
- Another challenge is that probabilistic models can be computationally intensive and require significant resources to develop and implement.

FAQs in Probabilistic Models in Machine

Q. What are some examples of probabilistic models?

Ans – Bayesian networks, Gaussian mixture models, hidden Markov models, and probabilistic graphical models are some most common example of probabilistic models.

Q. What are the limitation of probabilistic Models?

Ans – Probabilistic model assumes some underlying distribution and assumption in the parameters of data.

Q. What is difference between Parameteric probabilistic model and non-paramateric probabilistic model.

Ans – Parameteric distribution have some assumption about parameter of the distribution of the model whereas Non-prametric probabilistic does not have any assume about the distribution of the model.

Q. What are some advantages of Probalistic Model

Ans – Probalistic Model can easily handle noise and missing values in the data

3. Supervised Machine Learning

Discover what supervised machine learning is, how it compares to unsupervised machine learning and how some essential supervised machine learning algorithms work

Introduction

Supervised machine learning is a type of machine learning that learns the relationship between input and output. The inputs are known as features or 'X variables' and output is generally referred to as the target or 'y variable'. The type of data which contains both the features and the target is known as labeled data. It is the key difference between supervised and unsupervised machine learning, two prominent types of machine learning.

- What is Supervised Machine Learning
- Supervised vs. Unsupervised Machine Learning
- Semi-Supervised Machine Learning
- Supervised Machine Learning Algorithms:
- Linear Regression
- Decision Tree
- K Nearest Neighbors
- Random Forest
- Naive Bayes
- Supervised Machine Learning Python Code Example

What is Supervised Machine Learning?

Supervised machine learning learns patterns and relationships between input and output data. It is defined by its use of labeled data. A labeled data is a dataset that contains a lot of examples of Features and Target. Supervised learning uses algorithms that learn the relationship of Features and Target from the dataset. This process is referred to as Training or Fitting.

There are two types of supervised learning algorithms:

1. Classification
2. Regression

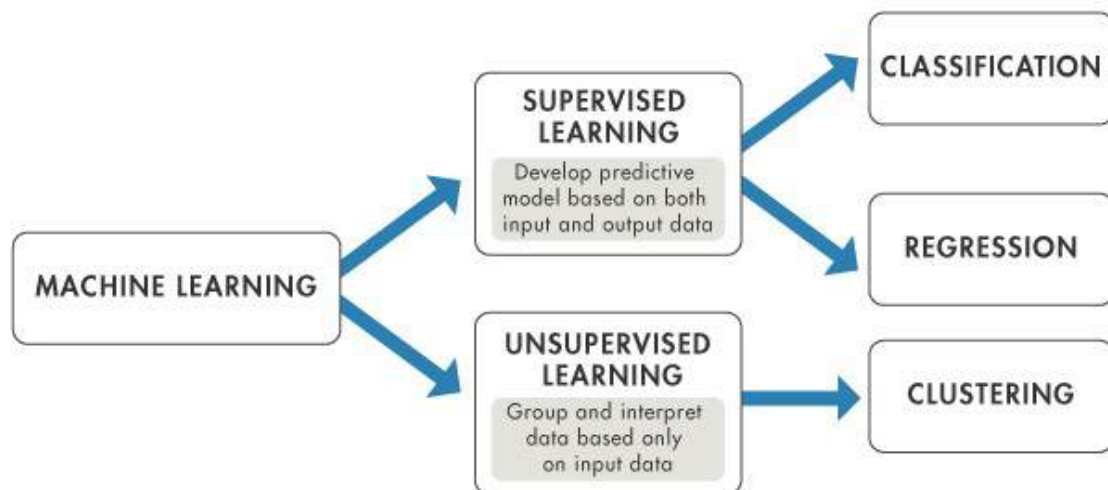


Image Source: <https://www.mathworks.com/help/stats/machine-learning-in-matlab.html>

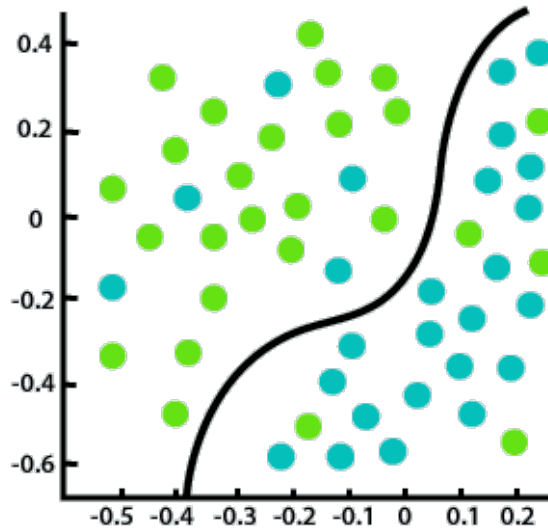
Classification Classification is a type of supervised machine learning where algorithms learn from the data to predict an outcome or event in the future. For example:

A bank may have a customer dataset containing credit history, loans, investment details, etc. and they may want to know if any customer will default. In the historical data, we will have Features and Target.

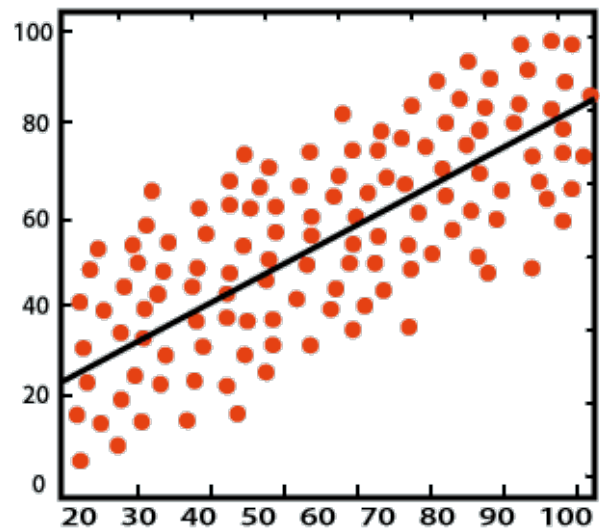
- Features will be attributes of a customer such as credit history, loans, investments, etc.
- Target will represent whether a particular customer has defaulted in the past (normally represented by 1 or 0 / True or False / Yes or No).

Classification algorithms are used for predicting discrete outcomes, if the outcome can take two possible values such as True or False, Default or No Default, Yes or No, it is known as Binary Classification. When the outcome contains more than two possible values, it is known as Multiclass Classification. There are many machine learning algorithms that can be used for classification tasks. Some of them are:

- Logistic Regression
- Decision Tree Classifier
- K Nearest Neighbor Classifier
- Random Forest Classifier
- Neural Networks



Classification



Regression

Regression Regression is a type of supervised machine learning where algorithms learn from the data to predict continuous values such as sales, salary, weight, or temperature. For example:

A dataset containing features of the house such as lot size, number of bedrooms, number of baths, neighborhood, etc. and the price of the house, a Regression algorithm can be trained to learn the relationship between the features and the price of the house.

There are many machine learning algorithms that can be used for regression tasks. Some of them are:

- Linear Regression
- Decision Tree Regressor
- K Nearest Neighbor Regressor
- Random Forest Regressor
- Neural Networks

Image Source: <https://static.javatpoint.com/tutorial/machine-learning/images/regression-vs-classification-in-machine-learning.png>

Supervised vs. Unsupervised Learning

Type of Data

The main difference between supervised and unsupervised machine learning is that supervised learning uses labeled data. Labeled Data is a data that contains both the Features (X variables) and the Target (y variable).

When using supervised learning, the algorithm iteratively learns to predict the target variable given the features and modifies for the proper response in order to "learn" from the training dataset. This process is referred to as Training or Fitting. Supervised learning models typically produce more accurate results than unsupervised learning but they do require human interaction at the outset in order to correctly identify the data. If the labels in the dataset are not correctly identified, supervised algorithms will learn the wrong details.

Unsupervised learning models, on the other hand, work in an autonomous manner to identify the innate structure of data that has not been labeled. It is important to keep in mind that validating the output variables still calls for some level of human involvement. For instance, an unsupervised learning model can determine that customers who shop online tend to purchase multiple items from the same category at the same time. However, a human analyst would need to check that it makes sense for a recommendation engine to pair Item X with Item Y.

Use-cases

There are two prominent use-cases for supervised learning i.e. Classification and Regression. In both the tasks a supervised algorithm learns from the training data to predict something. If the predicted variable is discrete such as "Yes" or "No", 1 or 0, "Fraud", or "No Fraud", then a classification algorithm is required. If the predicted variable is continuous like sales, cost, salary, temperature, etc., then the Regression algorithm is required.

Clustering and Anomaly Detection are two prominent use-cases in unsupervised learning.

Goals

The objective of supervised learning is to forecast results for new data based on a model that has learned from labeled training dataset. The kind of outcomes you can anticipate are known up front in the shape of labeled data. The objective of an unsupervised learning algorithm is to derive insights from massive amounts of data without explicit labels. Unsupervised algorithms also learn from the training dataset but the training data doesn't contain any labels.

Complexity

Supervised machine learning is straightforward relative to unsupervised learning. Unsupervised learning models generally require a large training set in order to yield the desired results, making them computationally complex.

Supervised Vs Unsupervised Learning, Explained

Supervised

X ₁	X ₂	X _p	Y

Target

Un-Supervised

X ₁	X ₂	X _p	Y

No Target

Image Source: <https://www.sharpsightlabs.com/blog/supervised-vs-unsupervised-learning/>

Semi-Supervised Machine Learning

Semi-supervised learning is a relatively new and less popular type of machine learning that, during training, blends a sizable amount of unlabeled data with a small amount of labeled data. Semi-supervised learning is between supervised learning (with labeled training data) and unsupervised learning (unlabeled training data).

Semi-supervised learning offers a lot of real-world applications. There is a dearth of labeled data in many fields. Because they involve human annotators, specialized equipment, or expensive, time-consuming studies, the labels (target variable) could be challenging to get.

Semi-supervised learning has two types:

1. transductive learning
2. inductive learning

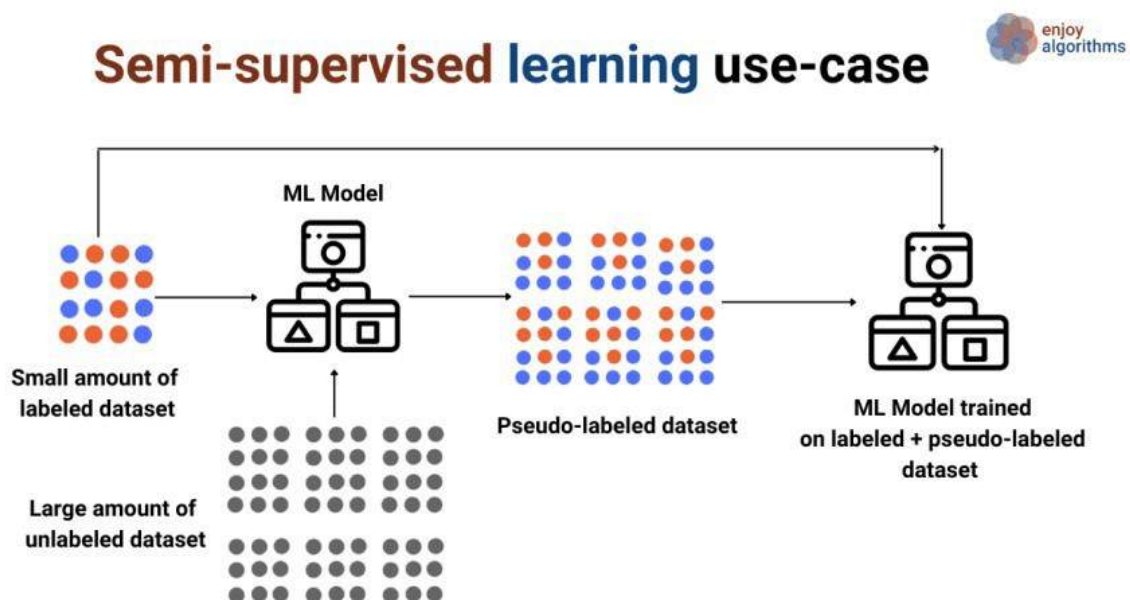


Image Source: <https://www.enjoyalgorithms.com/blogs/supervised-unsupervised-and-semisupervised-learning>

Supervised Machine Learning Algorithms

Linear Regression

Linear regression is one of the simplest machine learning algorithms available, it is used to learn to predict continuous value (dependent variable) based on the features (independent variable) in the training dataset. The value of the dependent variable which represents the effect, is influenced by changes in the value of the independent variable.

If you recall the “line of best fit” from school days, this is exactly what linear regression is. Predicting a person's weight based on their height is a straightforward example of this concept.

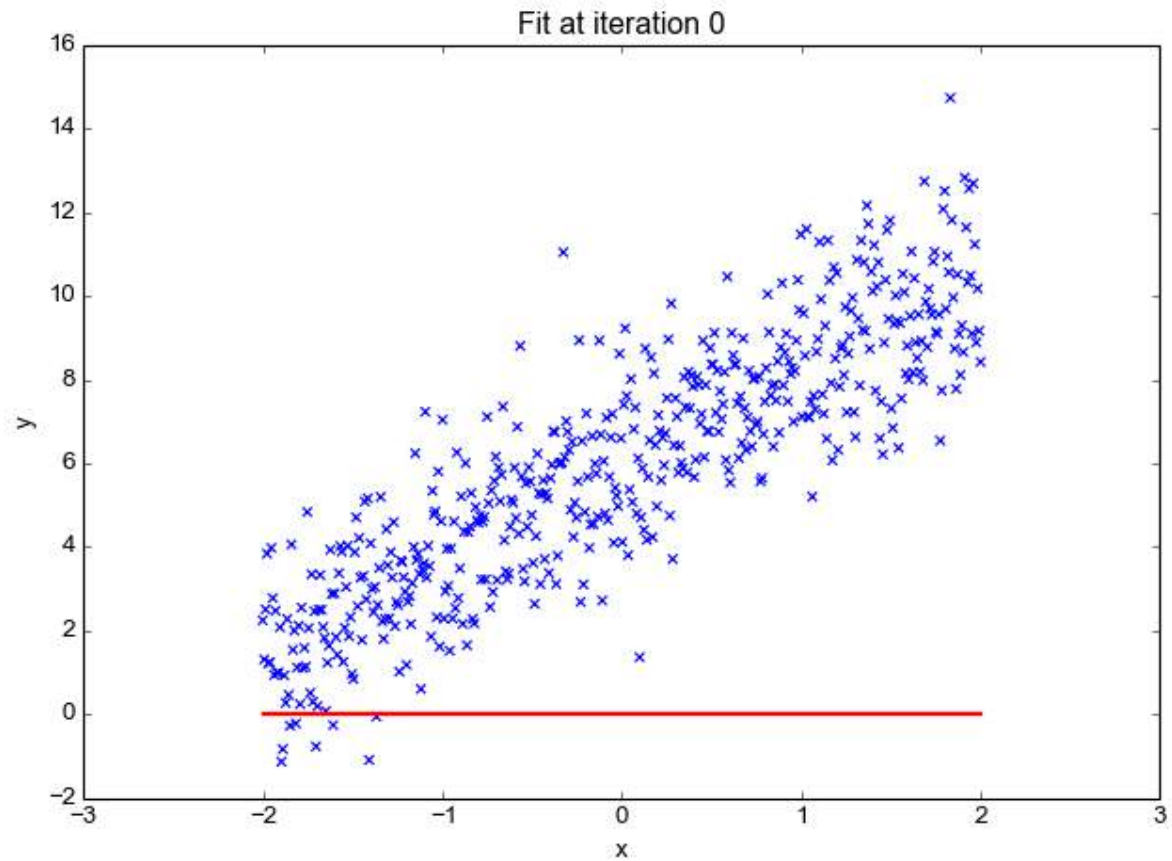


Image Source: http://primo.ai/index.php?title=Linear_Regression

PROS

Simple, easy to understand and interpret

Performs exceptionally well for linearly separable data

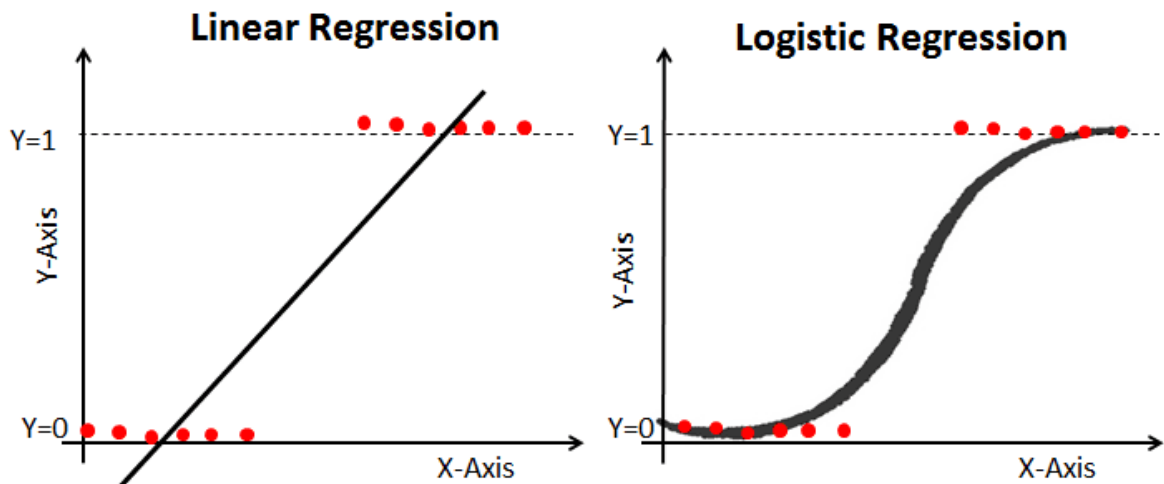
CONS

Easy to overfit

Assumes linearity between features and target variable.

Logistic Regression

Logistic Regression is a special case of Linear Regression where target variable (y) is discrete / categorical such as 1 or 0, True or False, Yes or No, Default or No Default. A log of the odds is used as the dependent variable. Using a logit function, logistic regression makes predictions about the probability that a binary event will occur.



PROS

Simple, easy to understand and interpret

Well-calibrated for output probabilities

CONS

Overfitting

Experience difficulty capturing complex relationships

Decision Tree

Decision Tree algorithms are a type of probability tree-like structural model that continuously separates data in order to categorize or make predictions depending on the results of the previous set of questions. The model analyzes the data and provides responses to the questions in order to assist you in making more informed choices.

You could, for instance, utilize a decision tree in which the answers Yes or No are used to select a certain species of bird based on data elements like the bird's feathers, its ability to fly or swim, the sort of beak it has, and so on.

TEST DATA

DECISION TREE

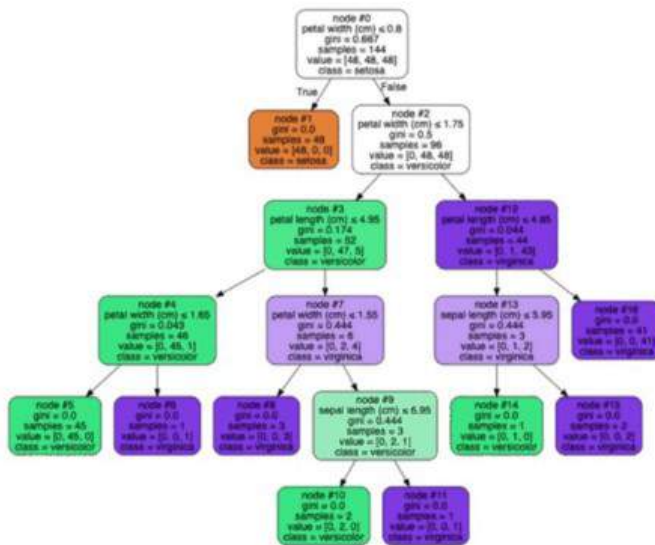


Image Source: <https://aigraduate.com/decision-tree-visualisation---quick-ml-tutorial-for-beginners/>

PROS

Very intuitive and easy to explain

CONS

Instable - a small change in training data can cause huge differences in prediction.

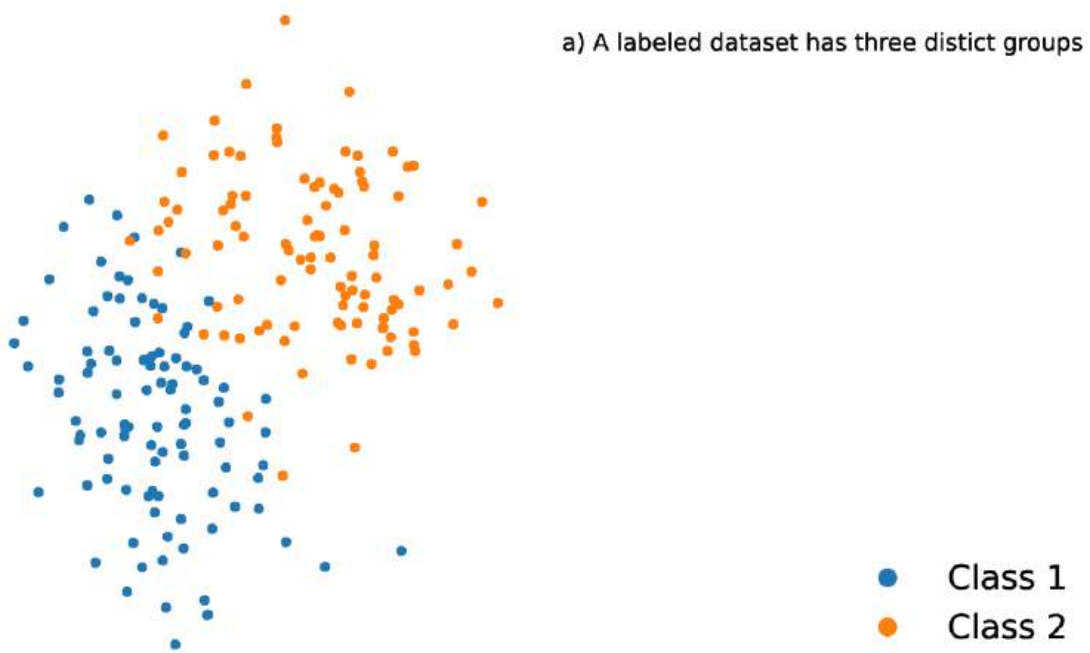
Decision trees don't require a lot of data preparation as some linear models require.

Prone to overfitting

K Nearest Neighbors

K-Nearest Neighbors is a statistical method that evaluates the proximity of one data point to another data point in order to decide whether or not the two data points can be grouped together. The proximity of the data points represents the degree to which they are comparable to one another.

For instance, suppose we had a graph with two distinct groups of data points that were located in close proximity to one another and named Group A and Group B, respectively. Each of these groups of data points would be represented by a point on the graph. When we add a new data point, the group of that instance will depend on which group the new point is closer to.



Source: <https://towardsdatascience.com/getting-acquainted-with-k-nearest-neighbors-ba0a9ecf354f>

PROS

Makes no assumption about the data

Intuitive and simple

CONS

Takes long time for training

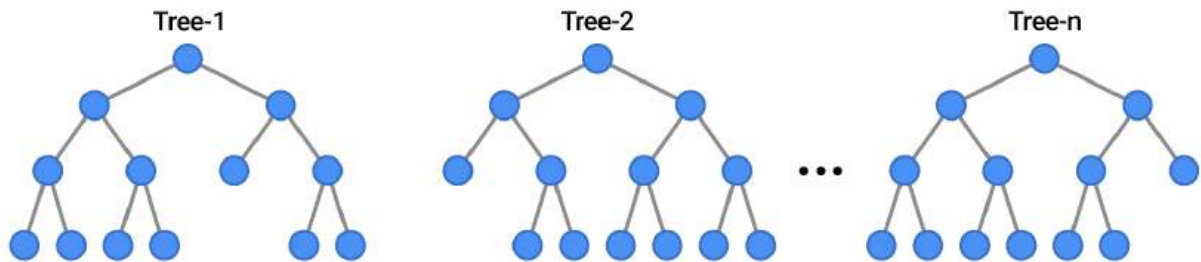
KNN works well with a small number of features but as the numbers of features grow it struggles to predict accurately.

Random Forest

Random Forest is another example of an algorithm that is built on trees, just like Decision Trees. In contrast to Decision Tree, which only consists of a single tree, Random Forest employs a number of Decision Trees in order to make judgments, creating what is essentially a forest of trees.

It does this by combining a number of different models in order to produce predictions, and it may be used for both classification and regression.

EXAMPLES



Source: <https://blog.tensorflow.org/2021/05/introducing-tensorflow-decision-forests.html>

PROS

Random Forests can handle non-linear relationships in the data easily.

Random Forests implicitly perform feature selection

CONS

Difficult to interpret because of multiple trees.

Random Forests are computationally expensive for large datasets.

Random Forests are computationally expensive for large datasets.

Naive Bayes The Bayes Theorem is a mathematical formula that is used for calculating conditional probabilities, and Naive Bayes is an application of that formula. The probability that an outcome will take place given that another event has already taken place is known as conditional probability.

It makes the prediction that the probabilities for each class belong to a specific class and that the class with the highest probability is the class that is regarded to be the class that is most likely to occur.

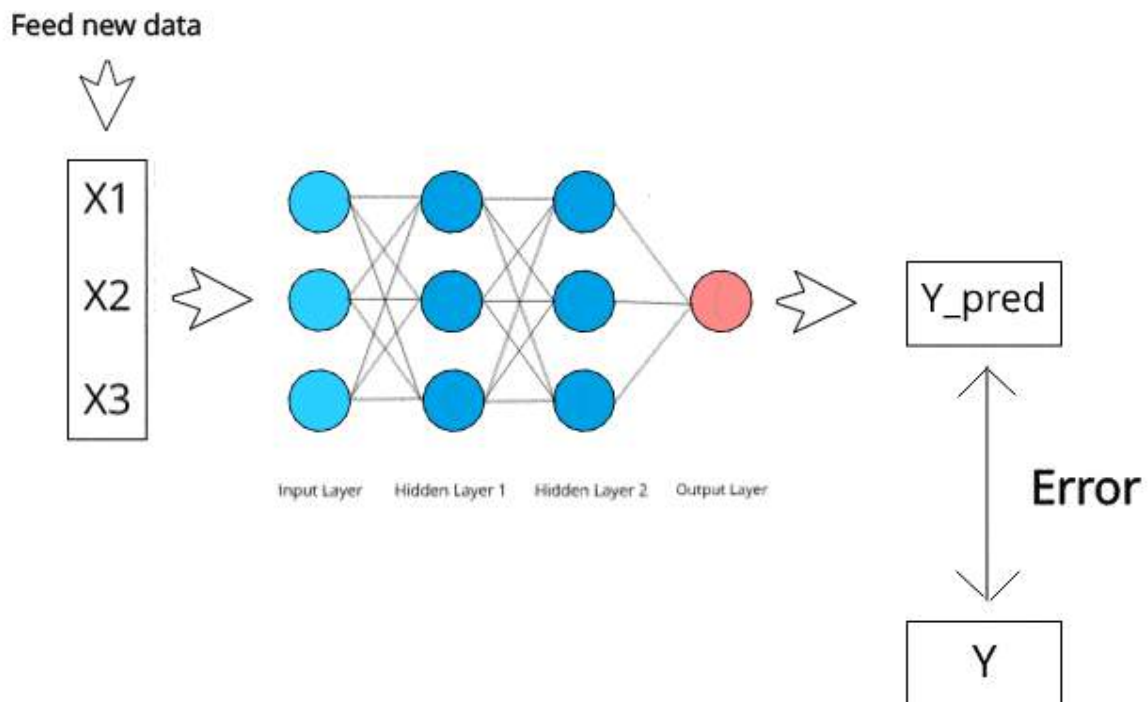


Image Source: <https://www.kdnuggets.com/2019/10/introduction-artificial-neural-networks.html>

PROS

The algorithm is very fast.
It is simple and easy to implement

CONS

It assumes all features are independent.
The algorithm encounters the "zero-frequency problem," where it gives a categorical variable with zero probability if its category was not present in the training dataset.

Supervised Machine Learning Python Code Example

In this part we will use [scikit-learn](#) in Python to train a Logistic Regression (Classification) model on a fake dataset. Check out the complete [Notebook](#) here.

```
# create fake binary classification dataset with 1000 rows and 10 features
from sklearn.datasets import make_classification

X, y = make_classification(n_samples = 1000, n_features = 10,
                          n_classes = 2)

# check shape of X and y
X.shape, y.shape
```

```
# train test split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.25, random_state=42)

# import and initialize logistic regression model
from sklearn.linear_model import LogisticRegression

lr = LogisticRegression()

# fit logistic regression model
lr.fit(X_train, y_train)

# generate hard predictions on test set
y_pred = lr.predict(X_test)
y_pred

# evaluate accuracy score of the model
from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_pred)

0.988
```

Machine learning has completely changed how we conduct business in recent years. A radical innovation that sets machine learning apart from other automation strategies is a departure from rules-based programming. Engineers may use data without explicitly training machines to solve problems in a certain way, thanks to supervised machine learning techniques.

In Supervised machine learning, the expected solution to a problem may not be known for future data, but may be known and captured in a historical dataset and the job of supervised learning algorithms is to learn that relationship from historical data to predict an outcome, event or a value in future.

Topic: Machine Learning in AI. Part 2

Unsupervised Learning

a. Unsupervised learning concept and workflow

Unsupervised learning is a branch of machine learning that deals with finding patterns and structure in unlabeled data.

Unlike supervised learning, where you have a predefined goal and a labeled dataset, unsupervised learning allows you to explore and discover the hidden features and relationships in the data without any guidance. This can be challenging, but also rewarding, as you can uncover new insights and generate novel solutions.

Supervised Learning	Unsupervised Learning
It uses known and labeled data as input	It uses unlabeled data as input
It has a feedback mechanism	It has no feedback mechanism
Model predicts the output	Model finds the hidden patterns in data

Supervised learning implies that model parameters are estimated in some way. Generally, the problem is reduced to minimize a cost function (or maximize some other metric) that depends directly on the parameters.

Ideally, cost functions are convex functions (on image) and the absolute minimum can be easily obtained following the descendant gradient. However, most practical cost functions are non-convex having multiple minimums, some local and some absolute. Non-convex functions are harder to optimize since algorithms cannot ensure finding a close solution.

If you remember a little bit from math courses, most of this implies calculating first and second derivatives, among other calculations — so yes, hard math underlies beneath all these concepts.

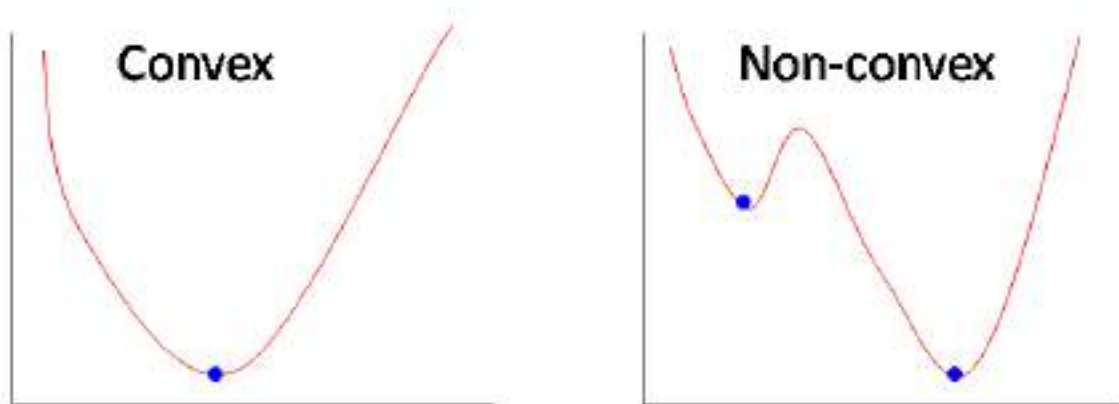


Image source: [Google Images](#)

Further on, unsupervised learning algorithms consist generally of iterative methods that depend on the initial conditions. These are great for this problems because after getting stuck in a local area early on, it can escape those to find the absolute minimum.

So, algorithms are mostly driven by the following steps:

1. Initialize parameters, either randomly or using some criteria.
2. Recalculate parameters following the proposed method.
3. Calculate metric, generally some likelihood-related value.
4. If metric increased its value from the past iteration, stop. If not, go to point 2.

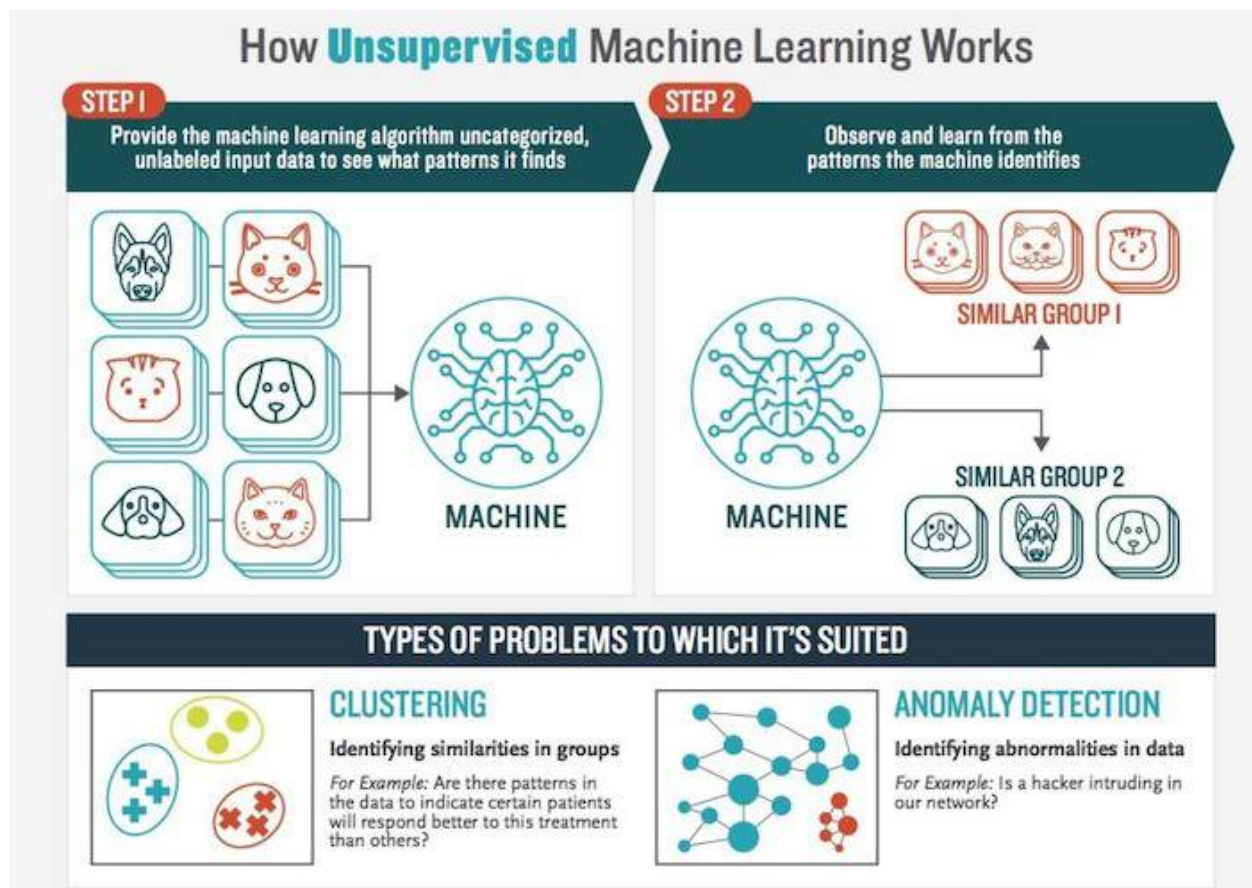


Image source: [Google Images](#)

Unsupervised learning is very powerful when having a really huge unlabeled set of data that needs to be processed in an efficient manner.

Unsupervised learning can be applied to a wide range of real-world problems, such as anomaly detection, customer segmentation, recommender systems, topic modeling, image compression, and more.

To apply unsupervised learning to these problems, you need to understand the problem domain, the data sources, the objectives, and the constraints.

You also need to preprocess and transform your data, such as scaling, normalization, encoding, or feature extraction, to make it suitable for unsupervised learning.

Finally, you need to interpret and communicate your results, such as visualizing the clusters or the reduced dimensions, or using them as inputs or outputs for other tasks or models.

Real-Life Applications of Unsupervised Learning

Market Basket Analysis

It is a machine learning model based on the algorithm that if you buy a certain group of items, you are less or more likely to buy another group of items.

Semantic Clustering

Semantically similar words share a similar context. People post their queries on websites in their own ways. Semantic clustering groups all these responses with the same meaning in a cluster to ensure that the customer finds the information they want quickly and easily. It plays an important role in information retrieval, good browsing experience, and comprehension.

Delivery Store Optimization

Machine learning models are used to predict the demand and keep up with supply. They are also used to open stores where the demand is higher and optimizing routes for more efficient deliveries according to past data and behavior.

Identifying Accident Prone Areas

Unsupervised machine learning models can be used to identify accident-prone areas and introduce safety measures based on the intensity of those accidents.

Types of unsupervised learning

Unsupervised learning can be broadly divided into two types: clustering (exploratory analysis) and dimensionality reduction.

Clustering

Clustering is the process of grouping similar data points together based on some similarity measure, such as distance or density. It is very useful in exploratory analysis because it can automatically identify patterns in data, and correlate them.

Clustering can help you:

- segment your data
- identify outliers
- find natural categories.

Example: If an analyst was trying to segment consumers, unsupervised clustering methods would be a great starting point. In situations where it is either impossible or impractical for a human to propose trends in the data, unsupervised learning can provide initial insights that can then be used to test individual hypotheses.

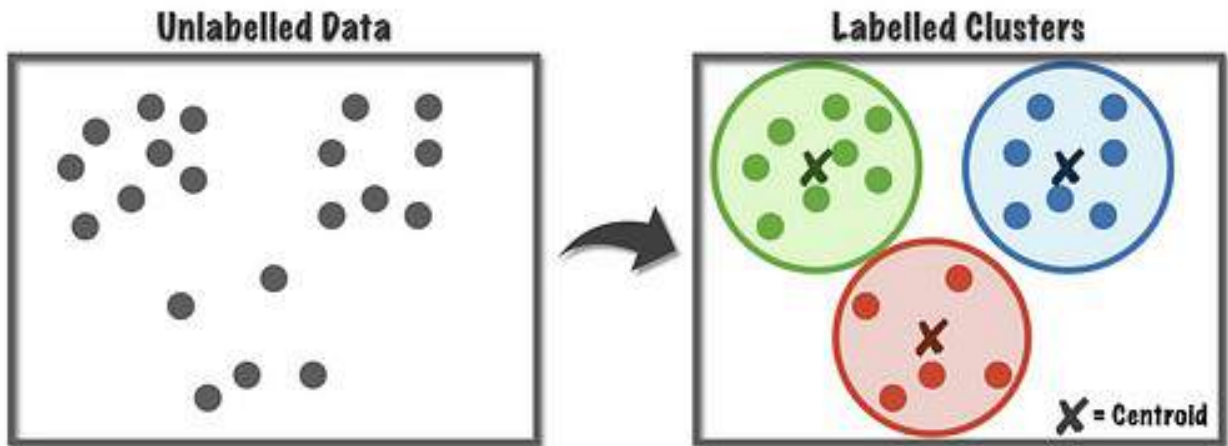


Image source: [Google Images](#)

For clustering, some of the common algorithms are:

- *k*-means
- hierarchical clustering
- DBSCAN
- Gaussian mixture models

Examples of use cases

- **Marketing:** It can be used to characterize & discover customer segments for marketing purposes.
- **Biology:** It can be used for classification among different species of plants and animals.
- **Libraries:** It is used for grouping different books on the basis of topics and information.
- **Insurance:** It is used to acknowledge the customers, their policies and identifying the frauds.
- **City Planning:** It is used to make groups of houses and to study their values based on their geographical locations and other factors present.
- **Earthquake studies:** By learning the earthquake-affected areas we can determine the dangerous zones.

Dimensionality reduction

Dimensionality is the number of variables, characteristics or features present in the dataset. In most cases, those columns are correlated and, therefore, there is some information that is redundant which increase the dataset's noise. This redundant information impacts negatively in Machine Learning model's training and performance and that is why using dimensionality reduction methods becomes important.

Dimensionality reduction is the process of reducing the number of features or variables in your data, while preserving as much information as possible. Dimensionality reduction can help you simplify your data, visualize it, and improve the performance of other machine learning models.

Dimensionality reduction is used as a data transformation technique for input to other machine learning methods such as classifications, or as a very efficient visualisation technique.

Dimensionality Reduction

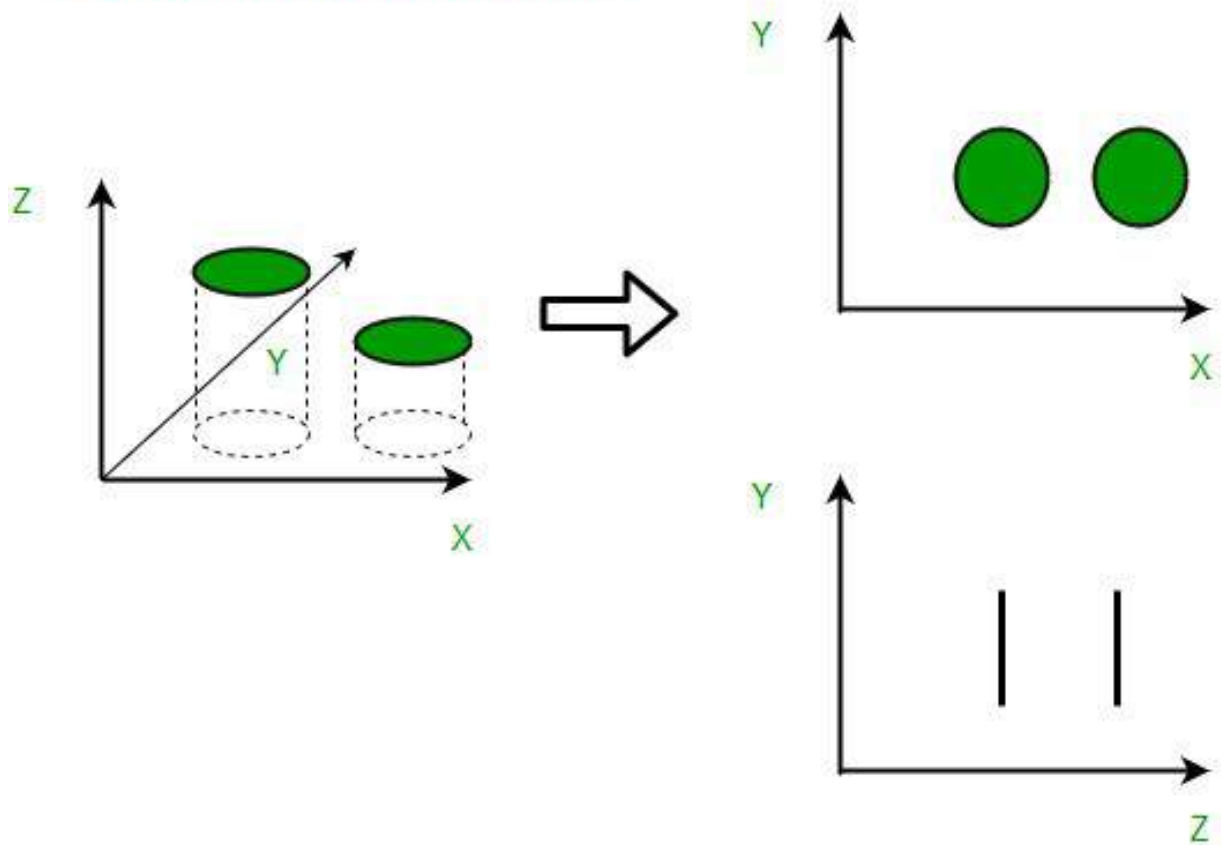


Image source: [Google Images](#)

For dimensionality reduction, some of the common techniques are:

- principal component analysis (PCA)
- singular value decomposition (SVD)
- t-distributed Stochastic Neighbor Embedding (t-SNE)
- autoencoders.

How do we know if it is working?

Unlike supervised learning, where you can measure the accuracy or error of your model against a ground truth, unsupervised learning does not have a clear way of evaluating and validating your results. However, there are some methods and metrics that can help you assess the quality and usefulness of your unsupervised learning outcomes.

Note 1: These links have a lot of mathematics :)

Note 2: Some of them require Medium account (free)

For clustering, you can use [internal metrics](#) to measure how well the clusters are separated and cohesive, such as:

- [silhouette score](#)
- [Davies-Bouldin index](#)
- [Calinski-Harabasz index](#)

You can also use external metrics to compare your clusters with some prior knowledge or labels, such:

- [adjusted Rand index](#)
- [normalized mutual information](#).

For dimensionality reduction, you can use metrics to measure how much information is retained or lost in the reduced data, such as:

- [explained variance ratio](#)
- [reconstruction error](#)
- [preservation of pairwise distances](#)

What algorithms to choose?

Depending on the type and characteristics of your data, you need to choose the appropriate algorithms and techniques for unsupervised learning. You should also consider the scalability, complexity, and interpretability of the algorithms and techniques, as well as their assumptions and limitations.

Most algorithms are suited for different purposes, while you are still starting your ML journey you can choose:

- For Clustering:
 - k-Means to group well-defined clusters. If you have too much data, you can use Mini Batch k -Means
 - DBSCAN to cluster groups with complex shapes
- For Dimensionality reduction
 - PCA to visualize linear data
 - T-SNE to visualize non-linear data

As a rule of thumb, always try multiple visualization and clustering techniques. Single technique can't tell you the whole story.

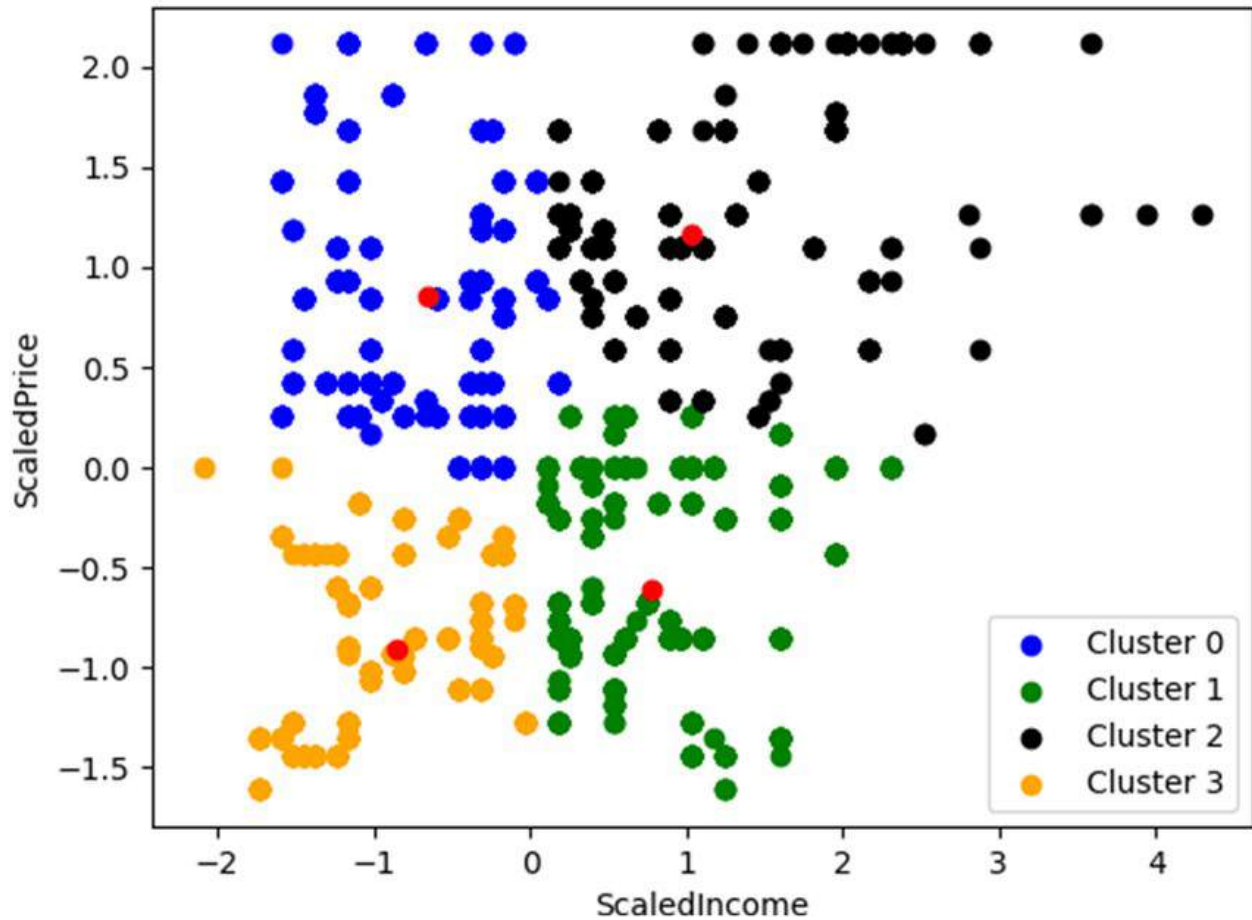
b. Clustering algorithms

Clustering also involves several different techniques.

k -Means Clustering

k -Means is an iterative algorithm that tries to partition the dataset into k pre-defined, distinct, non-overlapping subgroups or clusters, where each data point belongs to only one group.

The algorithm tries to make the intra-cluster data points as similar as possible while also keeping the clusters as different (*far*) as possible. It assigns data points to a cluster such that the sum of the squared distance between the data points and the cluster's centroid (*arithmetic mean of all the data points that belong to that cluster*) is minimized. The less variation we have within clusters, the more homogeneous the data points are within the same cluster.



Source: [Google Images](#)

The algorithm starts from a matrix of n features in p observations and proceeds as follows:

Initialisation: randomly assign the p observations to k clusters.

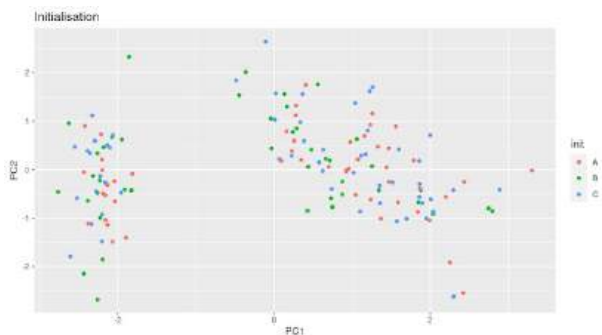


Image Source: [Google Images](#)

Iteration:

1. Calculate the centre of each subgroup as the average position of all observations in that subgroup.
2. Each observation is then assigned to the group of its nearest centre.

It's also possible to stop the algorithm after a certain number of iterations, or once the centres move less than a certain distance.

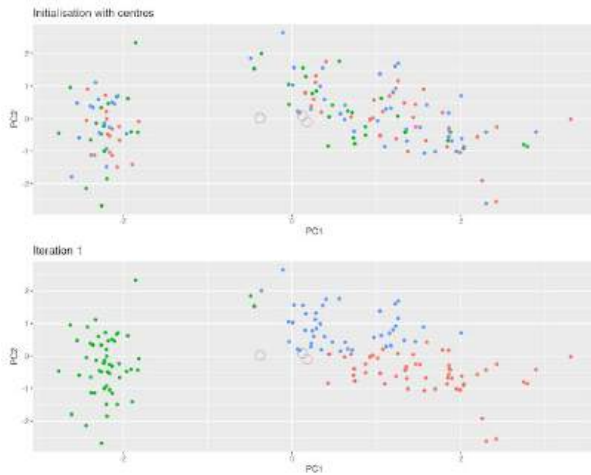


Image Source: [Google Images](#)

Termination: Repeat the iteration steps until no point changes its cluster membership.

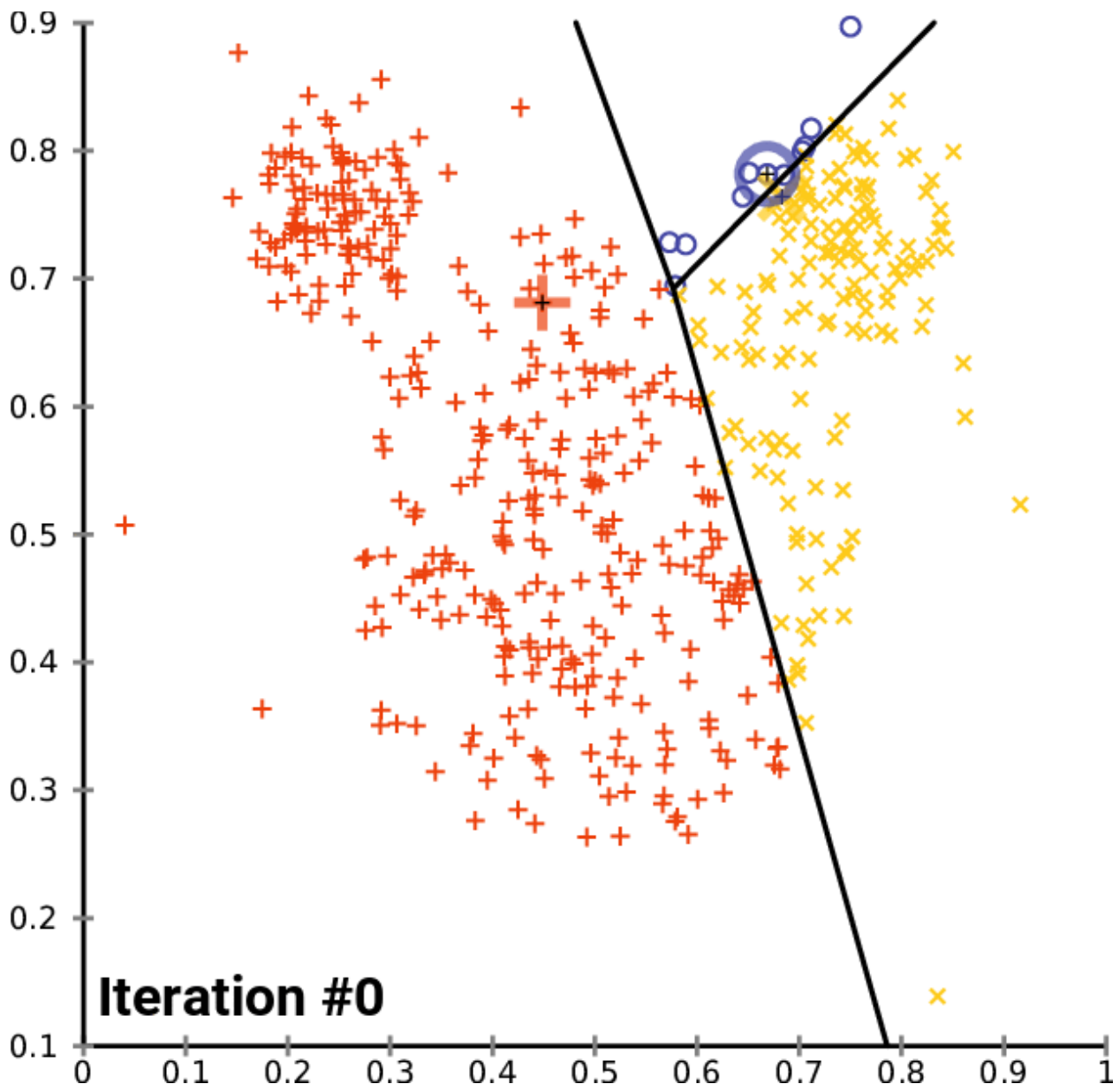


Image source: **Wikipedia**

Model selection

Due to the random initialisation, one can obtain different clustering results.

When k -means is run multiple times, the best outcome, i.e. the one that generates the smallest total within cluster sum of squares, is selected. The total within sum of squares is calculated as:

- For each cluster results:
 - for each observation, determine the squared euclidean distance to centre of cluster
 - sum all distances

Note that this is a local minimum; there is no guarantee to obtain a global minimum.

How to determine the number of clusters

- Run k-means with $k=1, k=2, \dots, k=n$
- Record total within SS for each value of k .
- Choose k at the *elbow* position, as illustrated below.

k -Means limitations

K-means clustering performs best on data that are spherical. Spherical data are data that group in space in close proximity to each other either. This can be visualized in 2 or 3 dimensional space more easily.

Data that aren't spherical or should not be spherical do not work well with k-means clustering. For example, k-means clustering would not do well on the below data:

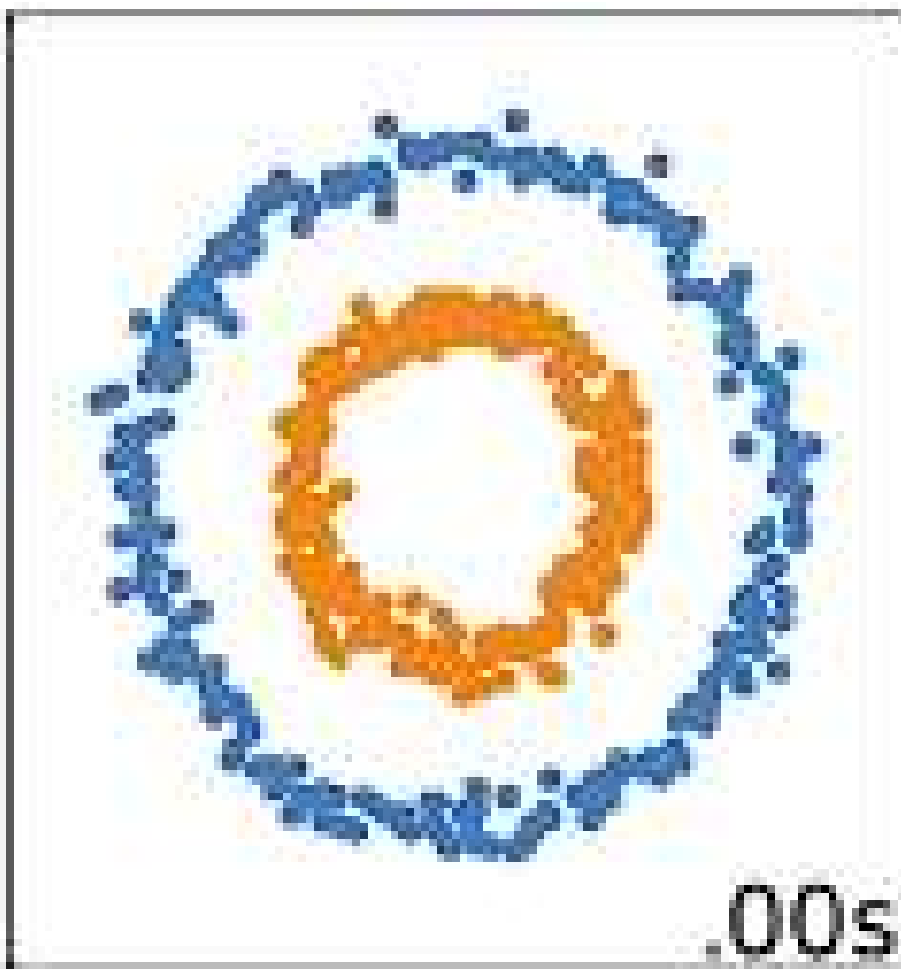


Image source: [Sklearn](#)

Hierarchical Clustering

Hierarchical clustering, also known as hierarchical cluster analysis (HCA), is an unsupervised clustering algorithm that can be categorized in two ways: agglomerative or divisive.

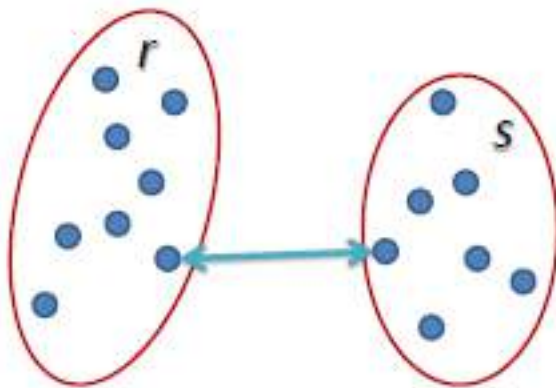
Agglomerative clustering is considered a "bottoms-up approach". In this method we assign each observation to its own cluster. Then, compute the similarity (e.g., distance) between each of the clusters and join the two most similar clusters. Finally, repeat these steps until there is only a single cluster left.

Divisive clustering can be defined as the opposite of agglomerative clustering; instead it takes a "top-down" approach. In this method we assign all of the observations to a single cluster and then partition the cluster to two least similar clusters. Finally, we proceed recursively on each cluster until there is one cluster for each observation.

Divisive clustering is not commonly used, but it is still worth noting in the context of hierarchical clustering.

Four different methods are commonly used to measure similarity:

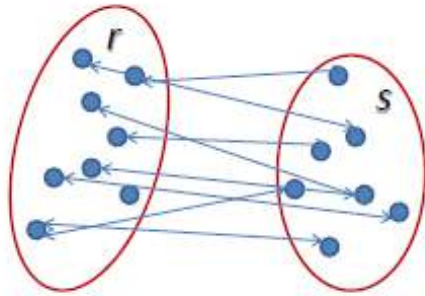
- *Single (or minimum) linkage*: This method is defined by the minimum distance between two points in each cluster. For example, the distance between clusters r and s is equal to the length of the arrow between their two closest points.



$$L(r, s) = \min(D(x_{ri}, x_{sj}))$$

Image Source: [Google Images](#)

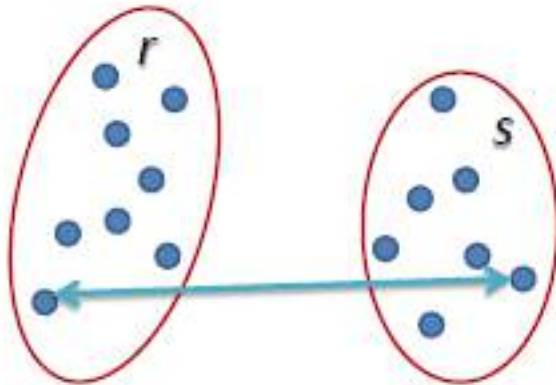
- *Average linkage*: This method is defined by the mean distance between two points in each cluster. For example, the distance between clusters r and s is equal to the average length of each arrow connecting the points of one cluster to the other.



$$L(r, s) = \frac{1}{n_r n_s} \sum_{i=1}^{n_r} \sum_{j=1}^{n_s} D(x_{ri}, x_{sj})$$

Image Source: [Google Images](#)

- *Complete (or maximum) linkage*: This method is defined by the maximum distance between two points in each cluster. For example, the distance between clusters r and s is equal to the length of the arrow between their two furthest points.



$$L(r, s) = \max(D(x_{ri}, x_{sj}))$$

Image Source: [Google Images](#)

- *Ward's linkage*: This method states that the distance between two clusters is defined by the increase in the sum of squared after the clusters are merged.

[Euclidean distance](#) is the most common metric used to calculate these distances; however, other metrics, such as [Manhattan distance](#), are also used.

These clustering processes are usually visualized using a dendrogram. [Dendograms](#) are tree diagrams frequently used to illustrate the arrangement of the clusters produced by hierarchical

clustering. The [clades](#) are arranged according to how similar (or dissimilar) they are. Clades that are close to the same height are similar to each other; clades with different heights are dissimilar — the greater the difference in height, the more dissimilarity.

An example involving the famous Iris data set is shown below.

Image Source: [Wikipedia](#)

Algorithm

Initialisation: Starts by assigning each of the n points its own cluster

Iteration:

- Find the two nearest clusters, and join them together, leading to $n - 1$ clusters
- Continue the cluster merging process until all are grouped into a single cluster

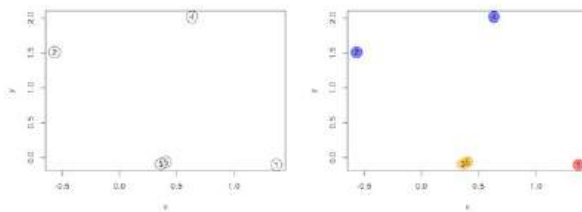


Image Source: [Google Images](#)

Termination: All observations are grouped within a single cluster.

After producing the hierarchical clustering result, we need to cut the tree (dendrogram) at a specific height to defined the clusters.

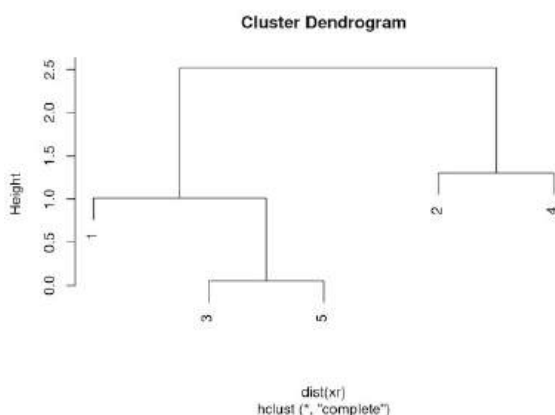


Image Source: [Google Images](#)

c. Dimensionality Reduction

While more data generally yields more accurate results, it can also impact the performance of machine learning algorithms (e.g., overfitting) and it can also make it difficult to visualize datasets. Dimensionality reduction is a technique used when the number of features, or dimensions, in a given dataset is too high. It reduces the number of data inputs to a manageable size while also preserving the integrity of the dataset as much as possible. It is commonly used in the preprocessing data stage, and there are a few different dimensionality reduction methods that can be used, such as:

Principal component analysis (PCA) is a type of dimensionality reduction algorithm which is used to reduce redundancies and to compress datasets through feature extraction. This method uses a linear transformation to create a new data representation, yielding a set of "principal components". The first principal component is the direction which maximizes the variance of the dataset. While the second principal component also finds the maximum variance in the data, it is completely uncorrelated to the first principal component, yielding a direction that is perpendicular, or orthogonal, to the first component. This process repeats based on the number of dimensions, where a next principal component is the direction orthogonal to the prior components with the most variance.

Singular value decomposition (SVD) is another dimensionality reduction approach which factorizes a matrix A into three, low-rank matrices. SVD is denoted by the formula

$$A = U \times \Sigma \times V^T,$$

where U and V are orthogonal matrices. Σ is a diagonal matrix, and Σ values are considered singular values of matrix A . Similar to PCA, it is commonly used to reduce noise and compress data, such as image files.

Autoencoders leverage neural networks to compress data and then recreate a new representation of the original data's input. Looking at the image below, you can see that the hidden layer specifically acts as a bottleneck to compress the input layer prior to reconstructing within the output layer. The stage from the input layer to the hidden layer is referred to as *encoding* while the stage from the hidden layer to the output layer is known as *decoding*.

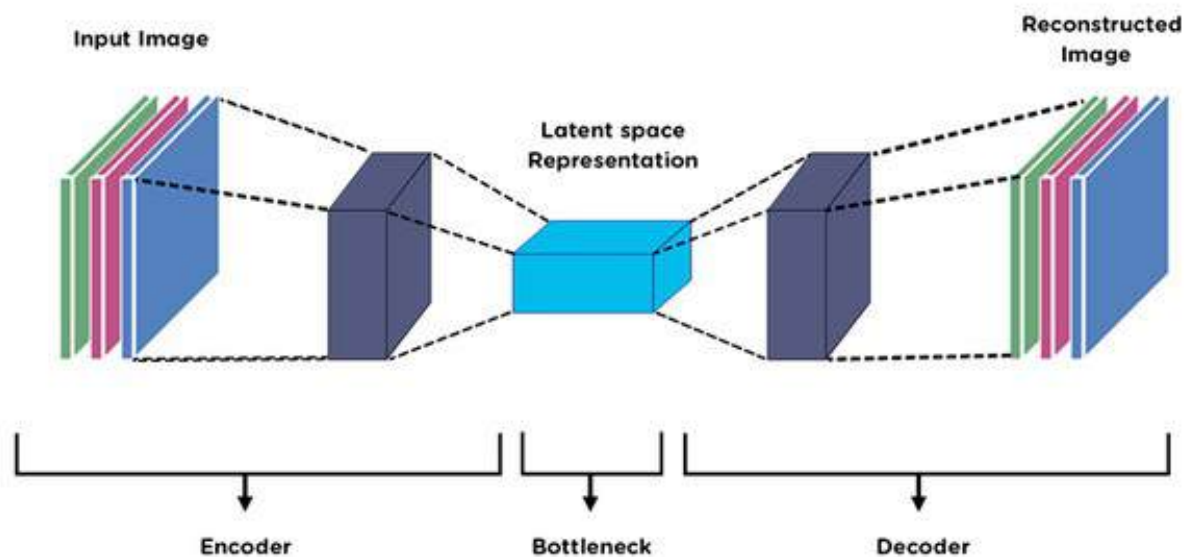


Image Source: [Google Images](#)

Principal Component Analysis (PCA)

Overview

Principal component analysis (PCA) is a statistical procedure that uses an [orthogonal transformation](#) to convert a set of observations of possibly correlated variables into a set of values of [linearly uncorrelated](#) variables called **principal components**. The number of distinct principal components is equal to the smaller of the number of original variables or the number of observations minus one.

This transformation is defined in such a way that the first principal component has the largest possible [variance](#) (that is, accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it is [orthogonal](#) to the preceding components. The resulting vectors are an uncorrelated [orthogonal basis set](#).

PCA is sensitive to the relative scaling of the original variables.

PCA was invented in 1901 by [Karl Pearson](#) as an analogue of the principal axis theorem in mechanics; it was later independently developed and named by [Harold Hotelling](#) in the 1930s.

Computing the [singular value decomposition \(SVD\)](#) is now the standard way to calculate a principal components analysis from a data matrix

Algorithm

For example, we have a small dataset of genes x and y .

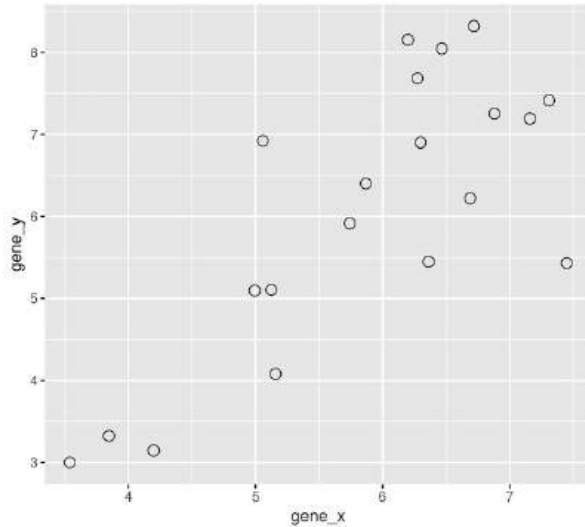


Image Source: [Google Images](#)

The goal of dimensionality reduction is to reduce the number of dimensions in a way that the new data remains useful.

One way to reduce a 2-dimensional data is by projecting the data onto a line. We can project our data on the x and y axes. These are called *linear projections*.

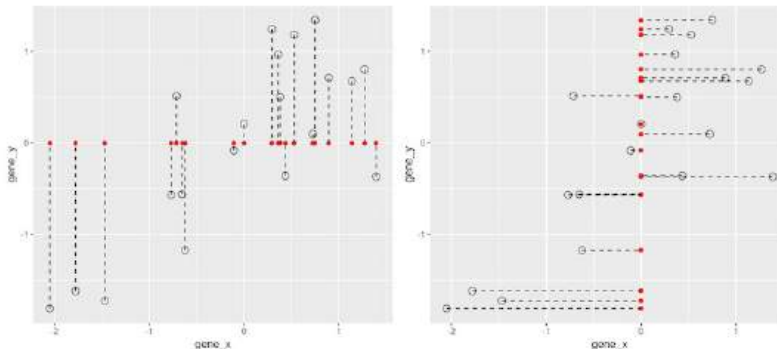


Image Source: [Google Images](#)

In general, we lose information when reducing the number of dimensions (e.g., from **2** (*plane*) to **1** (*line*)).

In the first example above (left), we lose all the measurements of gene y. In the second example (right), we lose all the measurements of gene x.

The goal of dimensionality reduction is to limit this loss.

We already know about linear regression. We can regress y onto x (*left*) and x onto y (*right*). These regression lines give us an approximate linear relationship between the expression of genes x and y. The relationship differs depending on which gene we choose to be the predictor and the response.

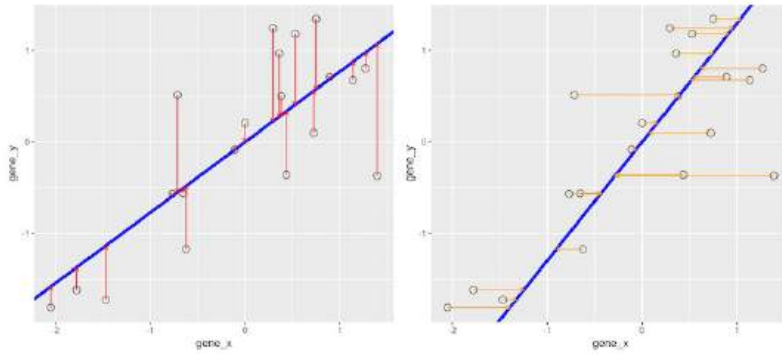


Image Source: [Google Images](#)

We now want a line that minimises distances in both directions. This line, called principal component, is also the ones that maximises the variance of the projections along itself.

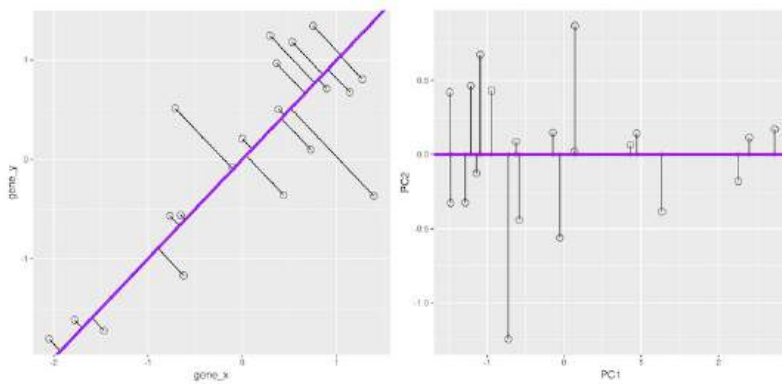


Image Source: [Google Images](#)

The second principal component is then chosen to be orthogonal to the first one. In our case, there is only one possibility.

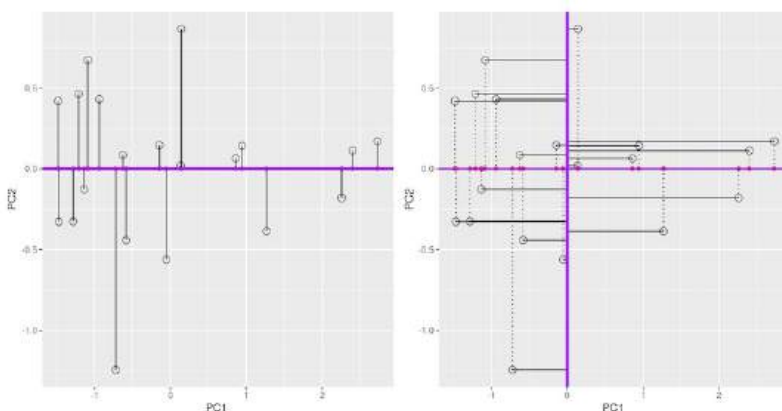


Image Source: [Google Images](#)

In the example above the variance along the PCs are 1.77 and 0.23, respectively. The first PC gets 88.6% of that variance, and the second PC gets merely 11.4%. This is also reflected in the different scales along the x and y axis.

To account for these differences in variation along the different PCs, it is better to represent a PCA plot as a rectangle, using an aspect ratio that is illustrative of the respective variances.

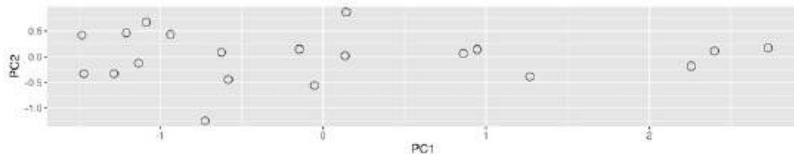


Image Source: [Google Images](#)

Principal components are linear combinations of the variables that were originally measured, they provide a new coordinate system. The PC in the previous example is a linear combination of `gene_x` and `gene_y`, more specifically

$$PC = c_1 gene_x + c_2 gene_y$$

It has coefficients (c_1, c_2) , also called loading.

PCA in general will find linear combinations of the original variables. These new linear combinations will maximise the variance of the data.

t-distributed Stochastic Neighbor Embedding (t-SNE)

t-SNE is a tool for data visualization. It reduces the dimensionality of data to 2 or 3 dimensions so that it can be plotted easily by converting distances between data in the original space to probabilities. Local similarities are preserved by this embedding.

It is used in data exploration and for visualizing high-dimension data. In simpler terms, t-SNE gives you a feel or intuition of how the data is arranged in a high-dimensional space.

It is an unsupervised, non-linear technique. Non-linear means that the algorithm allows us to separate data that cannot be separated by a straight line.

The t-SNE algorithm finds the similarity measure between pairs of instances in higher and lower dimensional space. After that, it tries to optimize two similarity measures. It does all of that in three steps.

1. t-SNE models a point being selected as a neighbor of another point in both higher and lower dimensions. It starts by calculating a pairwise similarity between all data points in the high-dimensional space using a Gaussian kernel. The points that are far apart have a lower probability of being picked than the points that are close together.
2. Then, the algorithm tries to map higher dimensional data points onto lower dimensional space while preserving the pairwise similarities. It is achieved by minimizing the divergence between the probability distribution of the original high-dimensional and lower-dimensional. The algorithm uses gradient descent to minimize the divergence. The lower-dimensional embedding is optimized to a stable state.
3. The optimization process allows the creation of clusters and sub-clusters of similar data points in the lower-dimensional space that are visualized to understand the structure and relationship in the higher-dimensional data.

Applications of t-SNE:

Apart from visualizing complex multi-dimensional data, t-SNE has other uses mostly in the medical field.

- **Clustering and classification:** to cluster similar data points together in lower dimensional space. It can also be used for classification and finding patterns in the data.
- **Anomaly detection:** to identify outliers and anomalies in the data.
- **Natural language processing:** to visualize word embeddings generated from a large corpus of text that makes it easier to identify similarities and relationships between words.
- **Computer security:** to visualize network traffic patterns and detect anomalies.
- **Cancer research:** to visualize molecular profiles of tumor samples and identify subtypes of cancer.
- **Geological domain interpretation:** to visualize seismic attributes and to identify geological anomalies.
- **Biomedical signal processing:** to visualize electroencephalogram (EEG) and detect patterns of brain activity.

One of the [popular t-SNE experiments](#) is to visualize 6000 handwritten digits (from 0 to 9) from MNIST dataset (image size is 28x28 pixel).



Image Source: [Google Images](#)

Result:

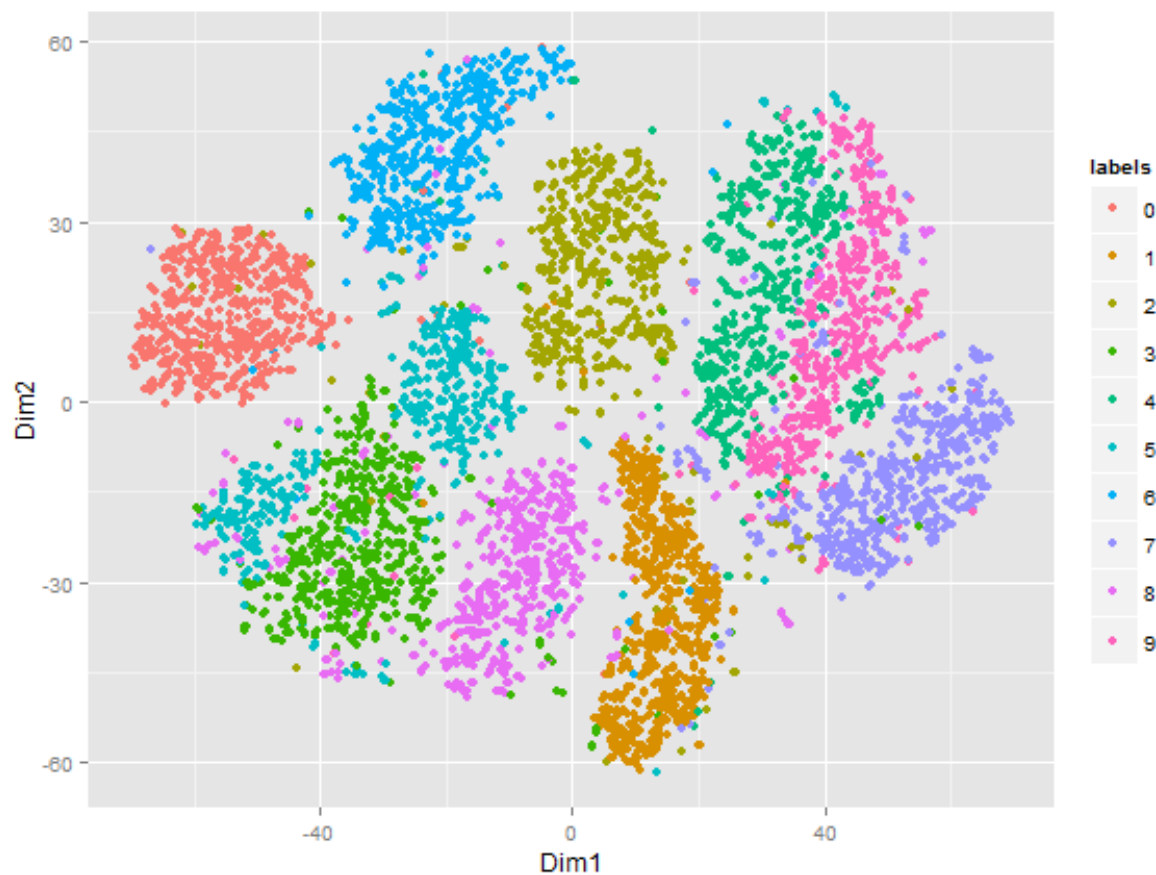


Image Source: [Google Images](#)

Hands-on: Applying unsupervised learning algorithms to real datasets

Install pre-requisites

```
%%capture
%pip install scikit-learn plotly pandas nbformat seaborn
```

k-Means

A good example of *k*-means that is used by Google and OpenAI is about analysing [California housing data from Kaggle](#).

We will cluster the houses by location and observe how house prices fluctuate across California.

The data includes 3 variables that we will use:

- **longitude**: A value representing how far west a house is. Higher values represent houses that are further West.

- **latitude**: A value representing how far north a house is. Higher values represent houses that are further north.
- **median_house_value**: The median house price within a block measured in USD.

```
from google.colab import drive
drive.mount('/content/drive')
%cd /content/drive/My Drive/Colab Notebooks/AI Course/

Mounted at /content/drive
/content/drive/My Drive/Colab Notebooks/AI Course

import pandas as pd

home_data = pd.read_csv('housing.csv', usecols = ['longitude',
'latitude', 'median_house_value'])
home_data.head()

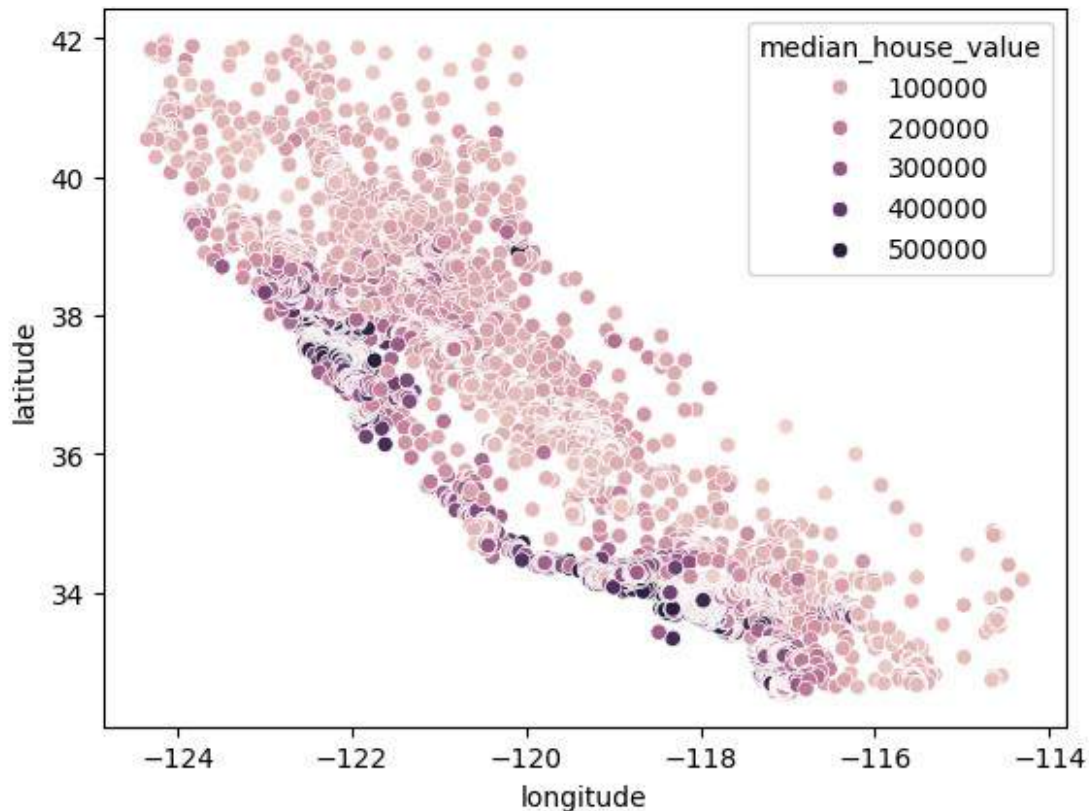
{"summary":{"\n  \"name\": \"home_data\",\n  \"rows\": 20640,\n  \"fields\": [\n    {\n      \"column\": \"longitude\",\n      \"properties\": {\n        \"dtype\": \"number\",\n        \"std\": 2.0035317235025882,\n        \"min\": -124.35,\n        \"max\": -114.31,\n        \"num_unique_values\": 844,\n        \"samples\": [\n          -118.63,\n          -119.86,\n          -121.26\n        ],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n      }\n    },\n    {\n      \"column\": \"latitude\",\n      \"properties\": {\n        \"dtype\": \"number\",\n        \"std\": 2.1359523974571153,\n        \"min\": 32.54,\n        \"max\": 41.95,\n        \"num_unique_values\": 862,\n        \"samples\": [\n          33.7,\n          34.41,\n          38.24\n        ],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n      }\n    },\n    {\n      \"column\": \"median_house_value\",\n      \"properties\": {\n        \"dtype\": \"number\",\n        \"std\": 115395.61587441387,\n        \"min\": 14999.0,\n        \"max\": 500001.0,\n        \"num_unique_values\": 3842,\n        \"samples\": [\n          194300.0,\n          379000.0,\n          230100.0\n        ],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n      }\n    }\n  ],\n  \"type\": \"dataframe\", \"variable_name\": \"home_data\"}
```

Let's start by visualizing the housing data. We look at the location data with a heatmap based on the median price in a block.

```
import seaborn as sns

sns.scatterplot(data = home_data, x = 'longitude', y = 'latitude', hue
= 'median_house_value')

<Axes: xlabel='longitude', ylabel='latitude'>
```

We can see that most of the expensive houses are on the west coast of California with different areas that have clusters of moderately priced houses.

Clusters are often easy to spot when you are only using 2 or 3 features. It becomes increasingly difficult or impossible when the number of features grows.

When working with distance-based algorithms, like k -Means Clustering, we must normalize the data. If we do not normalize the data, variables with different scaling will be weighted differently in the distance formula that is being optimized during training.

Example: if we were to include price in the cluster, in addition to latitude and longitude, price would have an outsized impact on the optimizations because its scale is significantly larger and wider than the bounded location variables.

```
from sklearn.model_selection import train_test_split
from sklearn import preprocessing

X_train, X_test, y_train, y_test =
train_test_split(home_data[['latitude', 'longitude']],
home_data[['median_house_value']], test_size=0.33, random_state=0)

X_train_norm = preprocessing.normalize(X_train)
X_test_norm = preprocessing.normalize(X_test)
```

For the first iteration, we will arbitrarily choose a number of clusters (k) of 3.

```
from sklearn.cluster import KMeans

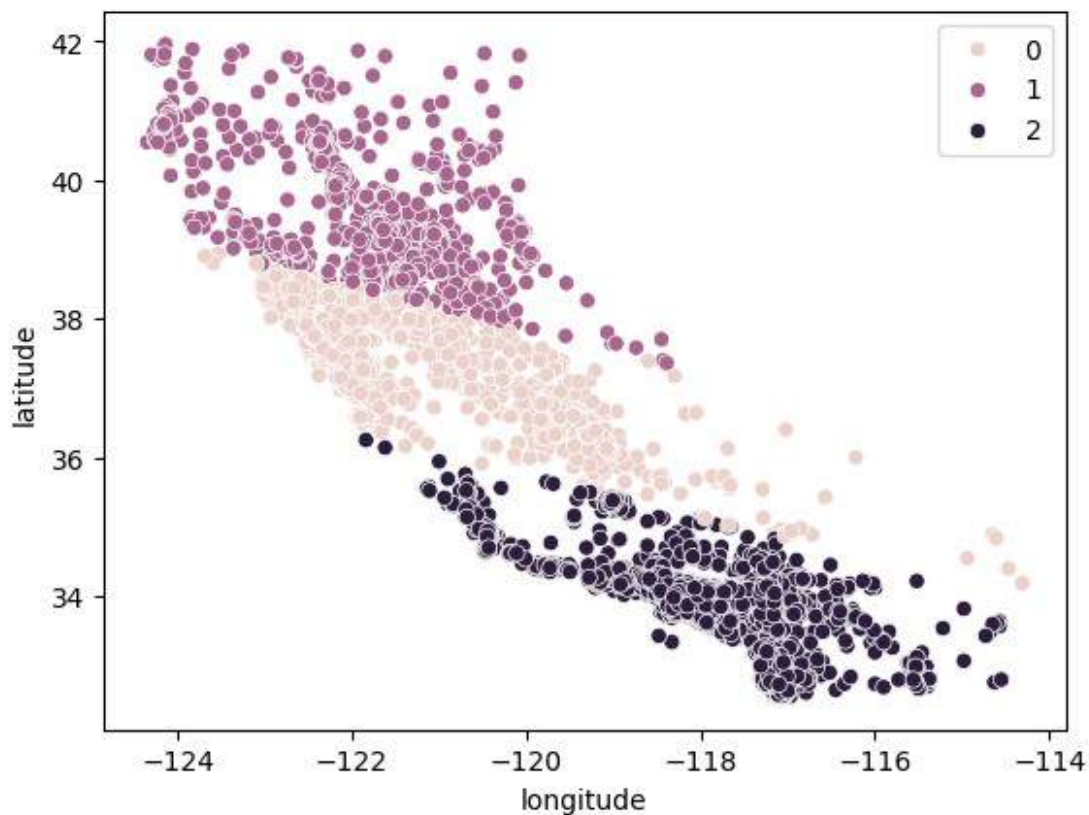
kmeans = KMeans(n_clusters = 3, random_state = 0, n_init='auto')
kmeans.fit(X_train_norm)

KMeans(n_clusters=3, n_init='auto', random_state=0)
```

Once the model is fit, we can visualize the data.

```
sns.scatterplot(data = X_train, x = 'longitude', y = 'latitude', hue =
kmeans.labels_)

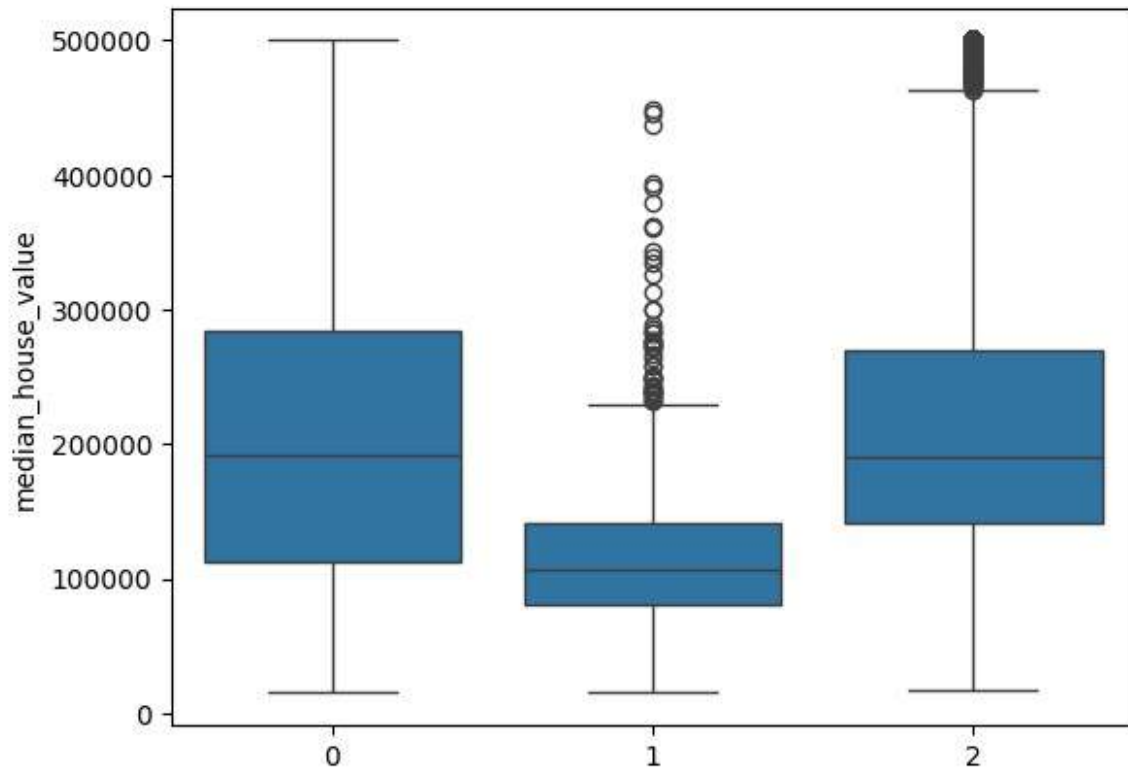
<Axes: xlabel='longitude', ylabel='latitude'>
```



We see that the data are now clearly split into 3 distinct groups (Northern California, Central California, and Southern California). We can also look at the distribution of median house prices in these 3 groups using a boxplot.

```
sns.boxplot(x = kmeans.labels_, y = y_train['median_house_value'])

<Axes: ylabel='median_house_value'>
```



We clearly see that the Northern and Southern clusters have similar distributions of median house values (clusters 0 and 2) that are higher than the prices in the central cluster (cluster 1).

We can evaluate performance of the clustering algorithm using a Silhouette score where a lower score represents a better fit. But since we have used only $k=3$, we do not know how good of a fit it is.

```
from sklearn.metrics import silhouette_score

silhouette_score(X_train_norm, kmeans.labels_, metric='euclidean')

0.7499371920917443
```

To figure out what is the best k we need to re-run the model with multiple values and see what is the best.

```
K = range(2, 8)
fits = []
score = []

for k in K:
    # train the model for current value of k on training data
    model = KMeans(n_clusters = k, random_state = 0,
n_init='auto').fit(X_train_norm)
```

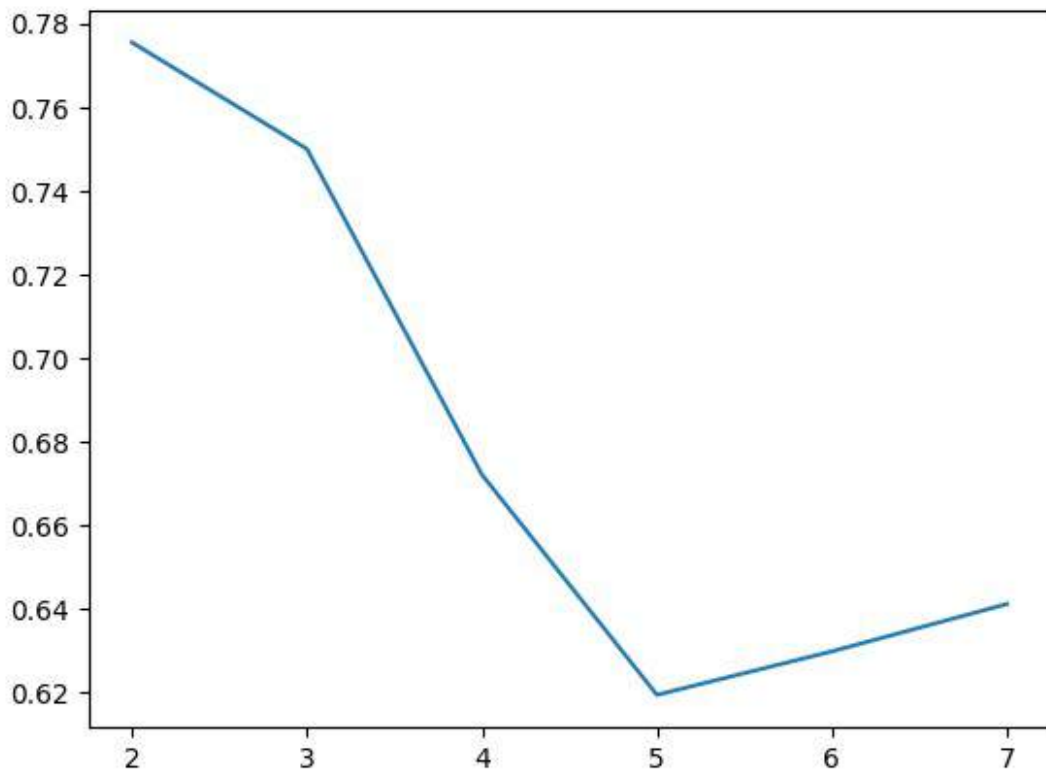
```
# append the model to fits
fits.append(model)

# Append the silhouette score to scores
score.append(silhouette_score(X_train_norm, model.labels_,
metric='euclidean'))
```

Typically, as we increase the value of k , we see improvements in clusters and what they represent until a certain point. Then we start to see diminishing returns or even worse performance.

```
sns.lineplot(x = K, y = score)
```

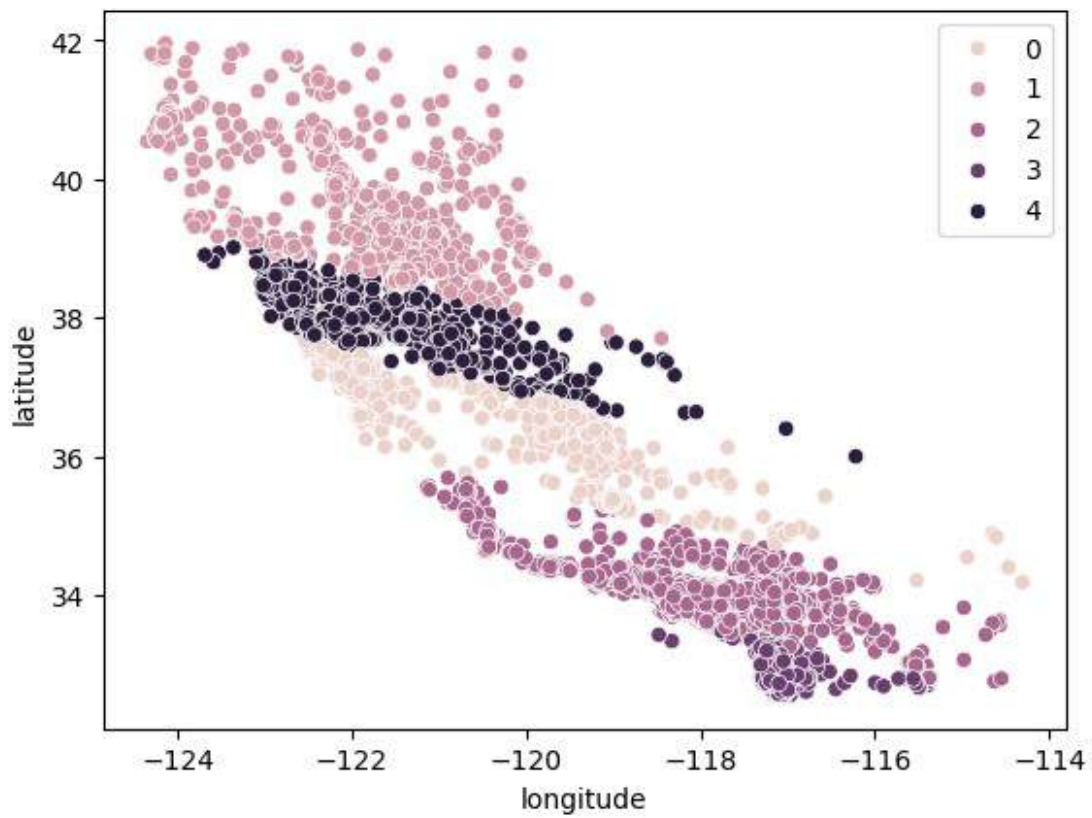
<Axes: >



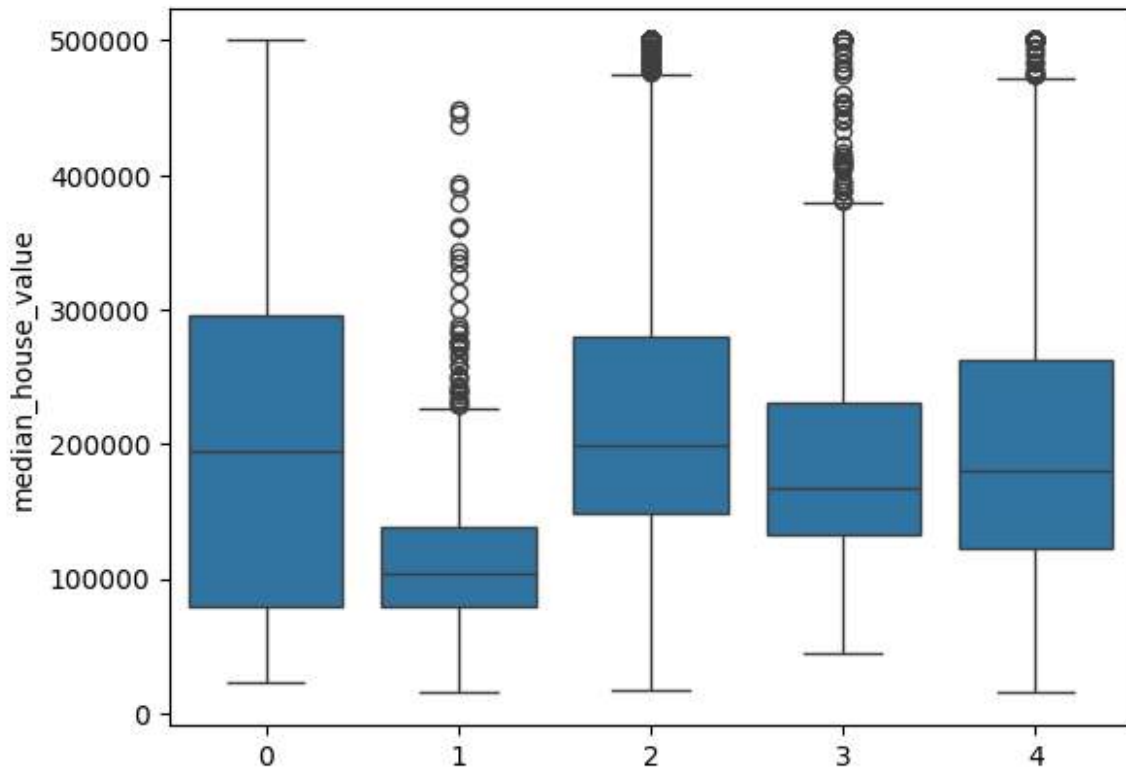
We see $k=5$ is probably the best we can do.

```
sns.scatterplot(data = X_train, x = 'longitude', y = 'latitude', hue =
fits[3].labels_)
```

<Axes: xlabel='longitude', ylabel='latitude'>



```
sns.boxplot(x = fits[3].labels_, y = y_train['median_house_value'])  
<Axes: ylabel='median_house_value'>
```



PCA & t-SNE

For this example we will generate synthetic data with 6 features, 1500 samples, and 3 classes.

First, we import required libraries.

```
import plotly.express as px
from sklearn.datasets import make_classification
```

Next, we'll generate and visualize the dataset.

```
X, y = make_classification(
    n_features=6,
    n_classes=3,
    n_samples=1500,
    n_informative=2,
    random_state=5,
    n_clusters_per_class=1,
)

fig = px.scatter_3d(x=X[:, 0], y=X[:, 1], z=X[:, 2], color=y,
    opacity=0.8)
fig.show()
```

Now we can apply the PCA algorithm on the dataset to return two PCA components. The `fit_transform` learns and transforms the dataset at the same time.

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
```

We can now visualize the results by displaying two PCA components on a scatter plot.

- **x**: First component
- **y**: Second companion
- **color**: target variable

```
fig = px.scatter(x=X_pca[:, 0], y=X_pca[:, 1], color=y)
fig.update_layout(
    title="PCA visualization",
    xaxis_title="First Principal Component",
    yaxis_title="Second Principal Component",
)
fig.show()
```

Let's apply the t-SNE algorithm to the dataset and compare the results.

```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, random_state=42)
X_tsne = tsne.fit_transform(X)
```

Similar to PCA, we will visualize two t-SNE components on a scatter plot.

```
fig = px.scatter(x=X_tsne[:, 0], y=X_tsne[:, 1], color=y)
fig.update_layout(
    title="t-SNE visualization",
    xaxis_title="First t-SNE",
    yaxis_title="Second t-SNE",
)
fig.show()
```

The result is quite better than PCA. We can clearly see three big clusters.

Machine Learning. Practical Tasks

For the completion of these tasks you may use [Kaggle](#) free datasets.

```
from google.colab import drive
drive.mount('/content/drive')
%cd /content/drive/My Drive/Colab Notebooks/SoftServe AI/
```

Task 1. Perform Basic Data Exploration on a Chosen Dataset

Problem: Exploring Datasets

Description: Exploring datasets is a crucial step in any data analysis or machine learning project. In this task, you will select a dataset from a reputable source, such as the UCI Machine Learning Repository, and conduct basic data exploration to gain insights into the dataset's structure, characteristics, and relationships between variables. This task will help you develop skills in data preprocessing, visualization, and summary statistics.

Steps:

1. **Dataset Selection:** Choose a dataset that interests you from the UCI Machine Learning Repository or a similar source. Make sure the dataset has a variety of attributes (columns) representing different aspects.
2. **Data Loading:** Load the dataset into your preferred programming environment (e.g., Python using Pandas).
3. **Data Inspection:** Explore the first few rows of the dataset using `head()` or `sample()` functions to get an initial sense of the data.
4. **Data Summary:** Use summary functions (e.g., `describe()`) to calculate basic statistics (mean, median, standard deviation) for numerical columns.
5. **Data Visualization:** Create visualizations such as histograms, scatter plots, and box plots to understand the distribution and relationships between different attributes.
6. **Missing Values:** Identify and handle missing values if any, using techniques like imputation or removal.
7. **Categorical Variables:** Analyze the distribution of categorical variables using bar plots or count plots.
8. **Correlation Analysis:** Calculate and visualize correlations between numerical attributes using techniques like correlation matrices or heatmaps.
9. **Data Insights:** Based on the data exploration, draw initial insights and observations. Identify potential outliers, trends, or interesting patterns.

Expected Output: A comprehensive analysis of the chosen dataset, including visualizations, summary statistics, and insights gained from the data exploration process. This exploration will provide a foundation for further data preprocessing, feature engineering, and model building.

Tips:

- Choose a dataset related to a domain you are curious about.
- Use libraries like Pandas, Matplotlib, and Seaborn for data manipulation and visualization.
- Document your findings and observations throughout the exploration process.

Learning Objectives: By completing this task, you'll gain practical experience in data exploration techniques, which are crucial for understanding and preparing data before applying machine learning algorithms. This task sets the stage for more advanced tasks such as data preprocessing and model building.

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

def load_dataset(file_path):
    """
    Load the dataset from the specified file path.

    Parameters:
    file_path (str): Path to the dataset file.

    Returns:
    pd.DataFrame: Loaded dataset as a Pandas DataFrame.
    """

    return dataset

def inspect_data(dataset, num_rows=5):
    """
    Display the first few rows of the dataset.

    Parameters:
    dataset (pd.DataFrame): Loaded dataset.
    num_rows (int): Number of rows to display.
    """

    return None

def calculate_summary_statistics(dataset):
    """
    Calculate summary statistics for numerical columns in the
    dataset.

    Parameters:
    dataset (pd.DataFrame): Loaded dataset.

    Returns:
```

```

    pd.DataFrame: Summary statistics.
    """

    return summary_stats

def visualize_data(dataset):
    """
    Create visualizations to explore the dataset.

    Parameters:
    dataset (pd.DataFrame): Loaded dataset.
    """
    # Example: Histograms for numerical columns

    # Example: Correlation heatmap

    return None

def handle_missing_values(dataset):
    """
    Handle missing values in the dataset.

    Parameters:
    dataset (pd.DataFrame): Loaded dataset.

    Returns:
    pd.DataFrame: Dataset with missing values handled.
    """
    # Example: Fill missing values with column mean

    return dataset_filled

# Load the dataset
file_path = 'organizations100.csv' # Replace with the actual file path
data = load_dataset(file_path)

# Perform data exploration
inspect_data(data)
summary_stats = calculate_summary_statistics(data)
visualize_data(data)
data_filled = handle_missing_values(data)

```

Task 2. Build a Binary Classifier for Predicting Binary Outcomes

Problem: Binary Classification

Description: Binary classification is a fundamental machine learning task where the goal is to build a model that can classify instances into one of two classes. In this task, you will develop a binary classifier using machine learning algorithms such as logistic regression or decision trees. The classifier will predict whether a given instance belongs to the positive class (e.g., spam) or the negative class (e.g., not spam).

Steps:

1. **Data Preparation:** Obtain a labeled dataset where instances are associated with binary class labels (0 or 1).
2. **Feature Extraction:** Preprocess and transform the raw data into numerical features suitable for model training. This may involve text tokenization, encoding categorical variables, and scaling numerical features.
3. **Model Selection:** Choose a binary classification algorithm to build the model. Examples include logistic regression, decision trees, support vector machines, and random forests.
4. **Model Training:** Split the dataset into training and validation sets. Train the selected model on the training data.
5. **Model Evaluation:** Evaluate the trained model's performance using appropriate evaluation metrics like accuracy, precision, recall, F1-score, and ROC-AUC.
6. **Hyperparameter Tuning:** Fine-tune the model's hyperparameters to optimize its performance.

Expected Output: A trained binary classifier that can predict binary outcomes for new instances. The model's performance metrics should demonstrate its accuracy and effectiveness in making correct predictions.

Tips:

- Start with a simple algorithm like logistic regression before exploring more complex models.
- Use libraries like scikit-learn to implement and train classifiers.
- Explore feature importance to understand which features contribute most to classification.

Learning Objectives: By completing this task, you'll gain hands-on experience in building and evaluating binary classifiers. You'll understand the workflow of preprocessing data, selecting appropriate algorithms, training models, and evaluating their performance using various metrics. These skills are essential for a wide range of applications, including spam detection, medical diagnosis, and fraud detection.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score, roc_auc_score

def load_dataset(file_path):
```

```

"""
Load a dataset from a CSV file.

This function reads a CSV file located at the specified
'file_path' and returns
the dataset as a pandas DataFrame.

Args:
    file_path (str): The path to the CSV file to be loaded.

Returns:
    pandas.DataFrame: The loaded dataset as a DataFrame.

"""

return dataset

def preprocess_text_data(dataset, target_column, vectorizer=None):
    """
    Preprocess text data for machine learning tasks.

    This function takes a dataset containing text features and their
    corresponding
    target values, and preprocesses the text data using the TF-IDF
    vectorization technique.
    It returns the transformed text features, target values, and the
    vectorizer used
    for transformation.

    Args:
        dataset (pandas.DataFrame): The dataset containing 'Text'
        and 'target_column'.
        target_column (str): The name of the column containing
        target values.
        vectorizer (sklearn.feature_extraction.text.TfidfVectorizer,
        optional):
            An optional TF-IDF vectorizer. If None, a new vectorizer
            will be created.

    Returns:
        tuple: A tuple containing:
            - text_features_transformed
            (scipy.sparse.csr.csr_matrix):
                Transformed text features using TF-IDF vectorization.
            - target (pandas.Series): Target values from the
            specified target_column.
            - vectorizer
            (sklearn.feature_extraction.text.TfidfVectorizer):
                The TF-IDF vectorizer used for transformation.

    """

    return text_features_transformed, target, vectorizer

```

```

def train_classifier(X_train, y_train):
    """
    Train a classifier using the provided training data.

    This function takes training data in the form of feature matrix
    'X_train' and
    corresponding target values 'y_train', and trains a logistic
    regression classifier.
    The trained classifier is returned for further use.

    Args:
        X_train (scipy.sparse.csr.csr_matrix or array-like):
        Training feature matrix.
        y_train (array-like): Target values for training.

    Returns:
        sklearn.linear_model.LogisticRegression: Trained logistic
        regression classifier.
    """

    return classifier

def evaluate_classifier(classifier, X_val, y_val):
    """
    Evaluate the performance of a classifier on validation data.

    This function takes a trained classifier, validation feature
    matrix 'X_val', and
    corresponding validation target values 'y_val'. It calculates
    and returns various
    performance metrics including accuracy, precision, recall, F1
    score, and ROC AUC score.

    Args:
        classifier (sklearn.base.BaseEstimator): Trained classifier
        to be evaluated.
        X_val (scipy.sparse.csr.csr_matrix or array-like):
        Validation feature matrix.
        y_val (array-like): Validation target values.

    Returns:
        dict: A dictionary containing performance metrics:
            - 'accuracy': Accuracy score.
            - 'precision': Precision score.
            - 'recall': Recall score.
            - 'f1_score': F1 score.
            - 'roc_auc': ROC AUC score.
    """

```

```

    return metrics

# Load the CyberBullyingTweets dataset
file_path = 'CyberBullying_Comments_Dataset.csv'
target_column = 'CB_Label' # Replace with the actual label column
                             name
data = load_dataset(file_path)

# Preprocess the text data and extract features
X_text, y, text_vectorizer = preprocess_text_data(data,
target_column)

# Split the data into train and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_text, y,
test_size=0.2, random_state=42)

# Train the logistic regression classifier
classifier = train_classifier(X_train, y_train)

# Evaluate the classifier
evaluation_metrics = evaluate_classifier(classifier, X_val, y_val)
print("Evaluation Metrics:", evaluation_metrics)

```

Task 3. House Price Prediction

In this task, your objective is to create a robust regression model capable of predicting house prices with accuracy. The model should utilize a comprehensive set of features such as square footage, number of bedrooms, and location to provide accurate and informative estimates of property values.

Your task involves the following steps:

1. **Data Collection and Preprocessing:** Obtain a dataset that includes relevant information about houses, including features like square footage, number of bedrooms, location, and other relevant attributes. Clean and preprocess the data to handle missing values, outliers, and ensure its suitability for training.
2. **Feature Selection and Engineering:** Identify the most influential features that contribute to accurate house price predictions. You may need to perform feature engineering to create new informative features or transform existing ones.
3. **Model Selection:** Choose an appropriate regression algorithm to build the predictive model. Linear regression, decision trees, random forests, or gradient boosting are common choices for this type of task.
4. **Training and Validation:** Split the dataset into training and validation subsets. Train the regression model on the training data and fine-tune its hyperparameters to achieve optimal performance.
5. **Model Evaluation:** Evaluate the trained model using appropriate regression evaluation metrics such as Mean Squared Error (MSE), Root Mean Squared Error

(RMSE), Mean Absolute Error (MAE), and R-squared. These metrics will provide insights into how well the model predicts house prices.

6. **Prediction:** Utilize the trained regression model to predict house prices for new or unseen data. This functionality can be integrated into a real-world application to assist potential buyers and sellers in making informed decisions.
7. **Fine-tuning:** If necessary, refine the model by adjusting hyperparameters, trying different algorithms, or incorporating more advanced techniques like regularization to improve its predictive performance.
8. **Documentation and Reporting:** Present your findings and insights from the analysis, including the chosen model, its performance metrics, and any observations regarding feature importance and trends.

The successful completion of this task will result in a reliable regression model that accurately predicts house prices based on key features, contributing to the real estate industry's ability to estimate property values effectively.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

def load_dataset(file_path):
    """
    Load a dataset from a CSV file.

    This function reads a CSV file located at the specified
    'file_path' and returns
    the dataset as a pandas DataFrame.

    Args:
        file_path (str): The path to the CSV file to be loaded.

    Returns:
        pandas.DataFrame: The loaded dataset as a DataFrame.
    """

    return dataset

def preprocess_data(dataset):
    """
    Preprocess the dataset by selecting relevant features and
    target.

    This function takes a dataset and extracts the 'square_footage'
    and 'num_bedrooms'
    columns as features, and the 'price' column as the target
    variable. The preprocessed
    features and target are returned for further use.
```

```

    Args:
        dataset (pandas.DataFrame): The dataset containing relevant
        columns.

    Returns:
        tuple: A tuple containing:
            - X (pandas.DataFrame): Preprocessed feature matrix with
            'square_footage' and 'num_bedrooms'.
            - y (pandas.Series): Target values from the 'price'
            column.

    """
    # Select relevant features and target

    return X, y

def train_regression_model(X_train, y_train):
    """
    Train a regression model using the provided training data.

    This function takes training data consisting of a feature matrix
    'X_train' and
    corresponding target values 'y_train', and trains a linear
    regression model.
    The trained model is returned for further use.

    Args:
        X_train (array-like): Training feature matrix.
        y_train (array-like): Target values for training.

    Returns:
        sklearn.linear_model.LinearRegression: Trained linear
        regression model.

    """

    return model

def evaluate_regression_model(model, X_val, y_val):
    """
    Evaluate the performance of a regression model on validation
    data.

    This function takes a trained regression 'model', validation
    feature matrix 'X_val',
    and corresponding validation target values 'y_val'. It
    calculates and returns various
    performance metrics including Mean Squared Error (MSE), Root
    Mean Squared Error (RMSE),
    Mean Absolute Error (MAE), and R-squared score.

```



```

    Args:
        model (sklearn.base.BaseEstimator): Trained regression model
        to be evaluated.
        X_val (array-like): Validation feature matrix.
        y_val (array-like): Validation target values.

    Returns:
        dict: A dictionary containing performance metrics:
            - 'MSE': Mean Squared Error.
            - 'RMSE': Root Mean Squared Error.
            - 'MAE': Mean Absolute Error.
            - 'R-squared': R-squared score.

    """

    return {'MSE': mse, 'RMSE': rmse, 'MAE': mae, 'R-squared':
r_squared}

# Load the dataset
file_path = 'House_prices.csv' # Replace with actual file path
data = load_dataset(file_path)

# Preprocess the data
X, y = preprocess_data(data)

# Split the data into train and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y,
test_size=0.2, random_state=42)

# Train the regression model
model = train_regression_model(X_train, y_train)

# Evaluate the model
evaluation_metrics = evaluate_regression_model(model, X_val, y_val)
print("Evaluation Metrics:", evaluation_metrics)

```

Task 4. K-Means Clustering: Exploring Similarity and Grouping

In this task, you will delve into the world of unsupervised machine learning by employing the K-Means clustering algorithm to uncover hidden patterns and groupings within a dataset. K-Means clustering is a fundamental technique that aims to partition data points into distinct clusters based on their similarity, providing valuable insights into data relationships without the need for labeled training data.

Problem Statement:

Your objective is to apply the K-Means clustering algorithm to a given dataset and gain a deeper understanding of its structure through cluster analysis. This process involves the following steps:

1. **Data Exploration and Preprocessing:** Begin by loading and exploring the dataset to understand its attributes and dimensions. Perform any necessary preprocessing, such as handling missing values, scaling features, or converting categorical variables if applicable.
2. **Choosing the Number of Clusters (K):** Determine the appropriate number of clusters (K) to partition the data. This step involves selecting a value for K that reflects the natural groupings within the data or leveraging techniques such as the elbow method or silhouette analysis.
3. **Feature Selection:** Depending on the dataset's characteristics, select the relevant features that will be used for clustering. K-Means operates based on feature similarity, so careful consideration of relevant features is crucial.
4. **Applying K-Means Clustering:** Utilize the K-Means algorithm to segment the data points into K clusters. The algorithm iteratively assigns data points to the nearest cluster center and recalculates the cluster centers based on the mean of the assigned data points.
5. **Visualizing Clusters:** After clustering, visualize the resulting clusters in two or three dimensions if possible. This step can provide insights into the spatial arrangement and relationships between different clusters.
6. **Interpreting Results:** Analyze and interpret the characteristics of each cluster. Identify commonalities among data points within a cluster and differences between clusters. This interpretation can provide valuable insights into the dataset's structure.
7. **Application and Insights:** Discuss the potential applications of the generated clusters in real-world scenarios. How can the identified groups be leveraged for decision-making or further analysis?
8. **Model Evaluation (Optional):** While K-Means is an unsupervised method and doesn't have a definitive evaluation metric, you can consider using metrics like silhouette score or inter-cluster distance to assess the quality of the clustering results.

By completing this task, you will gain proficiency in utilizing K-Means clustering as a tool to reveal underlying patterns and relationships within datasets. The insights gained from this analysis can provide valuable knowledge for various fields, including customer segmentation, image compression, and anomaly detection, among others.

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

def load_dataset(file_path):
    """
    Load a dataset from a CSV file.

    This function reads a CSV file located at the specified
```

```

'file_path' and returns
the dataset as a pandas DataFrame.

Args:
    file_path (str): The path to the CSV file to be loaded.

Returns:
    pandas.DataFrame: The loaded dataset as a DataFrame.
"""

return dataset

def preprocess_data(dataset, selected_features):
    """
    Scales selected features from the given dataset.

    This function applies standard scaling to the specified features
    in the dataset to normalize their values.

    Args:
        dataset (pandas.DataFrame): The dataset from which features
        are selected.
        selected_features (list): List of column names from the
        dataset to be scaled.

    Returns:
        numpy.ndarray: Scaled values of the selected features.
    """

    return X_scaled

def apply_kmeans_clustering(X_scaled, num_clusters):
    """
    Apply K-means clustering to scaled data.

    This function applies the K-means clustering algorithm to the
    scaled feature matrix
    'X_scaled' using the specified number of clusters
    'num_clusters'. It returns the
    cluster assignments for each data point and the cluster centers.

    Args:
        X_scaled (array-like): Scaled feature matrix for clustering.
        num_clusters (int): Number of clusters to form.

    Returns:
        tuple: A tuple containing:
        - clusters (array-like): Cluster assignments for each
        data point.
        - cluster_centers (array-like): Coordinates of cluster
        centers.

```

```

"""

    return clusters, kmeans.cluster_centers_

def visualize_clusters(X_scaled, clusters, cluster_centers):
    """
    Apply K-means clustering to scaled data.

    This function applies the K-means clustering algorithm to the
    scaled feature matrix
    'X_scaled' using the specified number of clusters
    'num_clusters'. It returns the
    cluster assignments for each data point and the cluster centers.

    Args:
        X_scaled (array-like): Scaled feature matrix for clustering.
        num_clusters (int): Number of clusters to form.

    Returns:
        tuple: A tuple containing:
            - clusters (array-like): Cluster assignments for each
            data point.
            - cluster_centers (array-like): Coordinates of cluster
            centers.

    """

    return None

# Load the dataset
file_path = 'features.csv' # Replace with actual file path
data = load_dataset(file_path)

# Selected features for clustering
selected_features = ['feature1', 'feature2'] # Replace with actual
feature names

# Preprocess the data
X_scaled = preprocess_data(data, selected_features)

# Choose the number of clusters (K)
num_clusters = 3 # Replace with the desired number of clusters

# Apply K-Means clustering
clusters, cluster_centers = apply_kmeans_clustering(X_scaled,
num_clusters)

# Visualize the clusters
visualize_clusters(X_scaled, clusters, cluster_centers)

```

Topic: Neural Networks in AI

Duration: 6 hours

Agenda:

1. Introduction to Artificial Neural Networks
 - a. Understanding the fundamentals of neural networks
 - b. Neurons and their connections in a neural network
 - c. Feedforward and backpropagation concepts
 - d. Real-world applications of neural networks in AI
2. Building Blocks of Neural Networks
 - a. Activation functions and their significance
 - b. Loss functions for different types of problems (e.g., regression, classification)
 - c. Optimization algorithms (e.g., Gradient Descent, Adam)
 - d. Regularization techniques (e.g., Dropout, L2 regularization)
3. Multi-layer Perceptrons
 - a. Understanding multi-layer perceptrons (MLPs)
 - b. Designing the architecture of an MLP
 - c. Forward propagation and backpropagation in MLPs
 - d. Hyperparameter tuning for better performance
4. Convolutional Neural Networks (CNNs)
 - a. Introduction to Convolutional Neural Networks
 - b. Convolution and pooling layers
 - c. Transfer learning and pre-trained CNNs
 - d. Applications of CNNs in image recognition and computer vision
5. Recurrent Neural Networks (RNNs)
 - a. Understanding Recurrent Neural Networks
 - b. Vanishing gradient problem and LSTM/GRU cells
 - c. Applications of RNNs in sequence data processing
 - d. Introduction to text generation using RNNs

6. Real-world Applications

- a. Use cases of neural networks in real-world scenarios (e.g., autonomous vehicles, medical diagnosis)

- b. Addressing bias and fairness in neural network models

7. Q&A and Wrap-up

- a. Answering participant questions and clarifications

- b. Recapitulation of the essential concepts covered during the course

- c. Encouraging further exploration of advanced neural network topics and AI applications

1. Introduction to Artificial Neural Networks

In this session, we will delve into the fascinating world of neural networks and explore their fundamental concepts, architecture, and real-world applications in AI.

Artificial Neural Networks, or ANNs, are computational models inspired by the human brain's neural structure. They consist of interconnected nodes called neurons, arranged in layers. The neurons receive inputs, process them, and pass the results to the next layer until the final output is generated.

An ANN is based on a collection of connected units or nodes called *artificial neurons*, which loosely model the *neurons* in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. An artificial neuron receives signals then processes them and can signal neurons connected to it. The "signal" at a connection is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs. The connections are called *edges*. Neurons and edges typically have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Neurons may have a threshold such that a signal is sent only if the aggregate signal crosses that threshold.

Typically, neurons are aggregated into layers. Different layers may perform different transformations on their inputs. Signals travel from the first layer (the input layer), to the last layer (the output layer), possibly after traversing the layers multiple times.

Neural networks learn (or are trained) by processing examples, each of which contains a known "input" and "result", forming probability-weighted associations between the two, which are stored within the data structure of the net itself. The training of a neural network from a given example is usually conducted by determining the difference between the processed output of the network (often a prediction) and a target output. This difference is the error. The network then adjusts its weighted associations according to a learning rule and using this error value. Successive adjustments will cause the neural network to produce output that is increasingly similar to the target output. After a sufficient number of these adjustments, the training can be terminated based on certain criteria. This is a form of supervised learning.

Such systems "learn" to perform tasks by considering examples, generally without being programmed with task-specific rules. For example, in image recognition, they might learn to identify images that contain cats by analyzing example images that have been manually labeled as "cat" or "no cat" and using the results to identify cats in other images. They do this without any prior knowledge of cats, for example, that they have fur, tails, whiskers, and cat-like faces. Instead, they automatically generate identifying characteristics from the examples that they process.

Feedforward is the process in which data flows through the network, layer by layer, from input to output. The activations of neurons are computed in a forward direction, leading to the final prediction.

Backpropagation is a crucial step in training neural networks. It involves propagating the error backward from the output layer to the input layer and adjusting the weights accordingly. This iterative process allows the network to learn from its mistakes and improve its predictions with each iteration.

The power of neural networks lies in their ability to learn from vast amounts of data and identify complex patterns. As a result, they have found wide-ranging applications in artificial intelligence:

1. **Image Recognition:** Neural networks have revolutionized image recognition tasks, enabling systems to identify objects, faces, and scenes with impressive accuracy.

2. **Natural Language Processing (NLP):** Neural networks have significantly improved NLP tasks like language translation, sentiment analysis, and text generation.
3. **Speech Recognition:** Voice-controlled devices and speech-to-text systems heavily rely on neural networks to understand and interpret human speech.
4. **Autonomous Vehicles:** Neural networks play a critical role in enabling self-driving cars to perceive and navigate their environments.
5. **Healthcare:** Neural networks have been used in medical diagnosis, drug discovery, and personalized treatment recommendations.

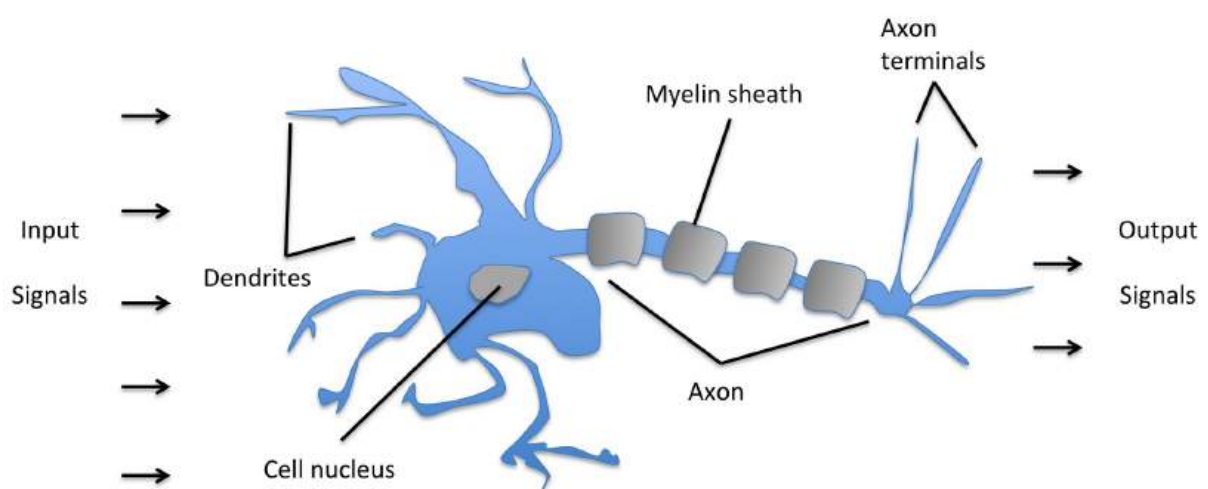
Let's get started and embark on this exciting journey into the world of Artificial Neural Networks!

2. Building Blocks of Neural Networks

- a. Activation functions and their significance
- b. Loss functions for different types of problems (e.g., regression, classification)
- c. Optimization algorithms (e.g., Gradient Descent, Adam)
- d. Regularization techniques (e.g., Dropout, L2 regularization)

2.1. The Formal Definition of an Artificial Neuron

Trying to understand how the biological brain works, in order to design artificial intelligence (AI), Warren McCulloch and Walter Pitts published the first concept of a simplified brain cell, the so-called McCulloch-Pitts (MCP) neuron, in 1943 (A Logical Calculus of the Ideas Immanent in Nervous Activity, W. S. McCulloch and W. Pitts, Bulletin of Mathematical Biophysics, 5(4): 115-133, 1943). Biological neurons are interconnected nerve cells in the brain that are involved in the processing and transmitting of chemical and electrical signals, which is illustrated in the following figure:



McCulloch and Pitts described such a nerve cell as a simple logic gate with binary outputs; multiple signals arrive at the dendrites, they are then integrated into the cell body, and, if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

Only a few years later, Frank Rosenblatt published the first concept of the perceptron learning rule based on the MCP neuron model (The Perceptron: A Perceiving and Recognizing Automaton, F. Rosenblatt, Cornell Aeronautical Laboratory, 1957). With his perceptron rule, Rosenblatt proposed an algorithm that would automatically learn the optimal weight coefficients that would then be multiplied with the input features in order to make the decision of whether a neuron fires (transmits a signal) or not. In the context of supervised learning and classification, such an algorithm could then be used to predict whether a new data point belongs to one class or the other.

An artificial neuron, also known as a **perceptron**, is a computational unit that receives one or more inputs, processes them, and produces an output. It is the fundamental building block of an artificial neural network.

Mathematically, an artificial neuron takes **input** features $x_1, x_2, \dots, x_n$, each associated with a **weight** $w_1, w_2, \dots, w_n$. The neuron then computes the weighted **sum** of the inputs and applies an **activation function** to produce the **output** y .

The weighted sum, often denoted as z , is given by: $z = w_1 \cdot x_1 + w_2 \cdot x_2 + \dots + w_n \cdot x_n$

Connections between neurons are represented by weights, which dictate the strength of communication between them. During training, these weights are adjusted through an optimization process, such as backpropagation, to minimize the difference between predicted outputs and actual outputs.

The activation function, denoted as $\phi(z)$, introduces non-linearity to the neuron and allows the neural network to model complex relationships between inputs and outputs.

The output y of the artificial neuron is the result of applying the activation function to the weighted sum z : $y = \phi(z)$

Commonly used activation functions include:

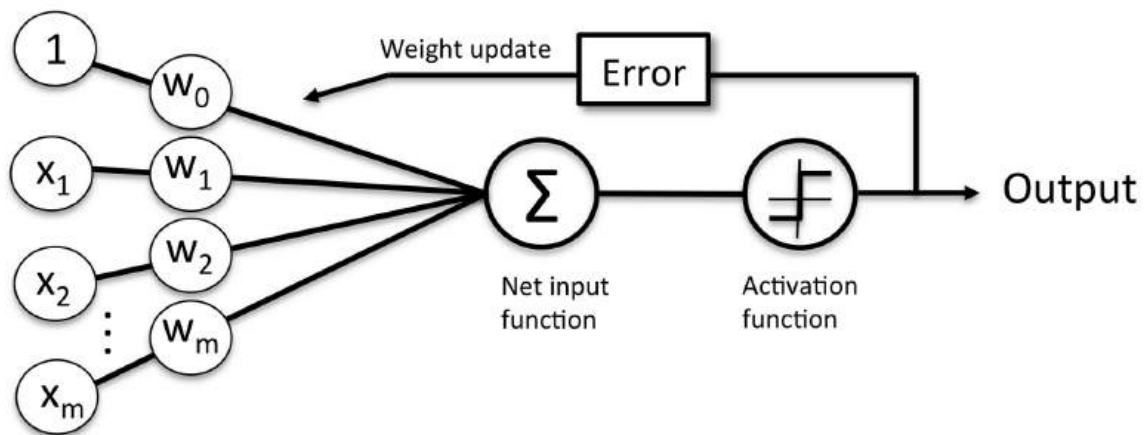
- Sigmoid function: $\phi(z) = \frac{1}{1 + e^{-z}}$
- ReLU (Rectified Linear Unit) function: $\phi(z) = \max(0, z)$
- Tanh (Hyperbolic Tangent) function: $\phi(z) = \frac{2}{1 + e^{-2z}} - 1$

The choice of the activation function depends on the specific task and the network's architecture.

Artificial neurons are designed to learn from data during the training process. The weights $w_1, w_2, \dots, w_n$ are initialized randomly, and as the neural network learns from training examples, these weights are adjusted to minimize the difference between predicted outputs and true outputs. This learning process is typically achieved using optimization algorithms like gradient descent.

Artificial neurons are organized into layers, such as the input layer, hidden layer(s), and output layer, to form an artificial neural network. The connections between neurons, represented by the weights, enable information flow and computation across the network.

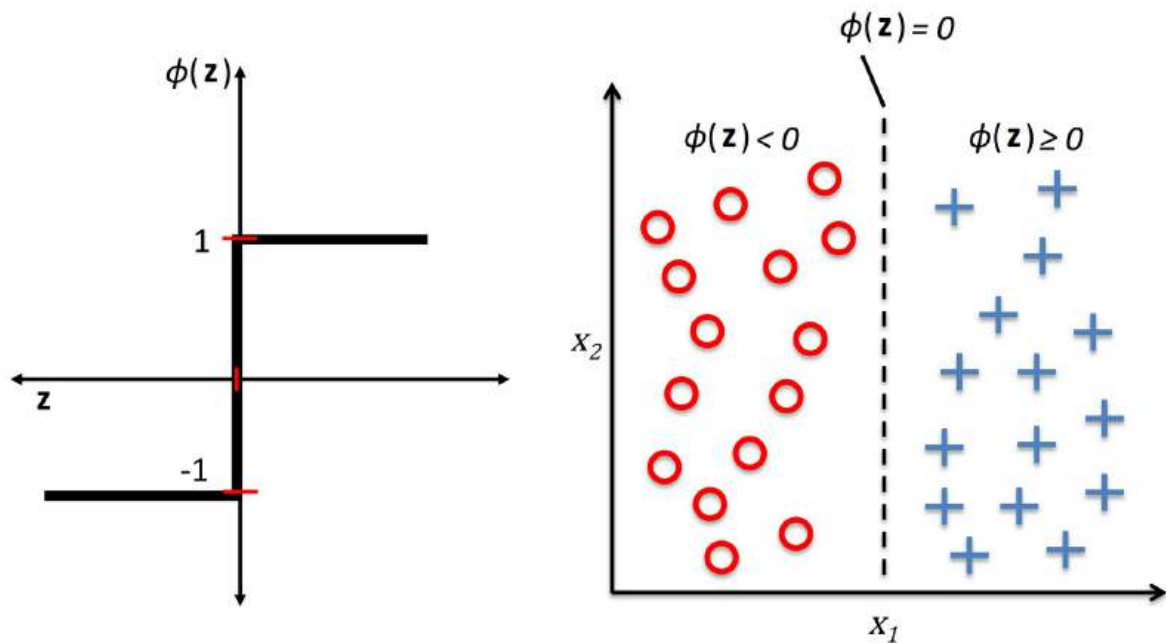
Now, before we jump into the implementation in the next section, what you just learned can be summarized in a simple diagram that illustrates the general concept of the perceptron:



The preceding diagram illustrates how the perceptron receives the inputs of an example, x , and combines them with the weights, w , to compute the net input. The net input is then passed on to the threshold function, which generates a binary output of -1 or $+1$ — the predicted class label of the example. During the learning phase, this output is used to calculate the error of the prediction and update the weights.

In summary, the formal definition of an artificial neuron involves taking inputs, computing the weighted sum of these inputs, applying an activation function, and producing an output. This mathematical formulation forms the basis of how neural networks learn and make predictions.

The following figure illustrates how the net input $z = w_1 \cdot x_1 + w_2 \cdot x_2 + \dots + w_n \cdot x_n$ is squashed into a binary output (-1 or 1) by the decision function of the perceptron (left subfigure) and how it can be used to discriminate between two linearly separable classes (right subfigure):



2.2. Activation Functions

Activation functions are an essential component of artificial neural networks. They introduce non-linearity to the network, enabling it to learn complex patterns and relationships in the data. Activation functions determine the output of a neuron based on the weighted sum of its inputs. The choice of activation function plays a critical role in the network's performance, convergence speed, and ability to solve different types of problems. Let's explore some common activation functions and their significance:

1. Sigmoid Function:

The sigmoid activation function maps the input to a range between 0 and 1. It has an S-shaped curve and is given by the formula:

$$\sigma(x) = 1 / (1 + \exp(-x))$$

Significance:

- Historically used as the primary activation function, especially in the output layer for binary classification problems.
- Provides a smooth gradient, making it suitable for backpropagation during training.
- The output is bounded between 0 and 1, which can be interpreted as probabilities.

2. ReLU (Rectified Linear Unit):

ReLU activation is defined as the maximum of 0 and the input value. It is one of the most popular activation functions for hidden layers in deep neural networks:

$$\text{ReLU}(x) = \max(0, x)$$

Significance:

- Fast to compute since it involves simple thresholding.
- Solves the vanishing gradient problem and helps prevent the "dying ReLU" problem.

- Contributes to faster convergence during training compared to sigmoid and tanh.

3. Tanh (Hyperbolic Tangent):

The tanh activation function is similar to the sigmoid but maps the input to a range between -1 and 1:

$$\tanh(x) = (2 / (1 + \exp(-2*x))) - 1$$

Significance:

- Provides a smoother gradient than the sigmoid function.
- Outputs are centered around zero, making it more suitable for zero-centered data.

4. Leaky ReLU:

Leaky ReLU is a variation of ReLU that allows a small, non-zero gradient for negative input values to avoid dead neurons:

$$\text{Leaky ReLU}(x) = \max(\alpha * x, x) \quad \text{where } \alpha \text{ is a small positive constant (e.g., 0.01)}$$

Significance:

- Helps mitigate the issue of dying ReLU, where neurons can get stuck during training with zero gradients.

5. Swish:

Swish is a newer activation function that combines ReLU and sigmoid-like behavior:

$$\text{Swish}(x) = x * \text{sigmoid}(x)$$

Significance:

- Performs well in various deep learning architectures and shows promising results.

The choice of the activation function depends on the nature of the problem, the data distribution, and the neural network architecture. ReLU and its variants (Leaky ReLU, Swish) are widely used due to their simplicity and effectiveness in deep learning. However, it's always important to experiment with different activation functions and evaluate their impact on the model's performance for a specific task.

2.3. Loss functions for different types of problems

Loss functions, also known as cost functions or objective functions, play a crucial role in training Neural Networks. They quantify how well the network is performing by comparing the predicted outputs to the true labels or target values. The goal of training is to minimize the loss function, which indicates that the network is making accurate predictions. The choice of the loss function depends on the type of problem being solved—regression or classification. Let's explore the common loss functions for each type:

1. Regression Problems:

In regression problems, the goal is to predict a continuous numerical value. Common loss functions for regression include:

- **Mean Squared Error (MSE):** MSE is one of the most widely used loss functions for regression tasks. It calculates the average of the squared differences between predicted values (y_{pred}) and true labels (y_{true}).

$$\text{MSE}(y_{\text{true}}, y_{\text{pred}}) = (1/n) * \sum (y_{\text{true}} - y_{\text{pred}})^2$$

where n is the number of data points.

- **Mean Absolute Error (MAE):** MAE is an alternative to MSE that calculates the average of the absolute differences between predicted values and true labels.

$$\text{MAE}(y_{\text{true}}, y_{\text{pred}}) = (1/n) * \sum |y_{\text{true}} - y_{\text{pred}}|$$

- **Huber Loss (Smooth L1 Loss):** Huber Loss is a combination of MSE and MAE and is less sensitive to outliers.

$$\begin{aligned} \text{Huber}(y_{\text{true}}, y_{\text{pred}}, \delta) &= \sum \{0.5 * (y_{\text{true}} - y_{\text{pred}})^2\} \text{ if } |y_{\text{true}} - y_{\text{pred}}| \leq \delta \\ &= \delta * |y_{\text{true}} - y_{\text{pred}}| - 0.5 * \delta^2 \text{ otherwise} \end{aligned}$$

where δ is a hyperparameter controlling the threshold for switching between MSE and MAE.

2. Classification Problems:

In classification problems, the goal is to predict the class or category to which a data point belongs. Common loss functions for classification include:

- **Binary Cross-Entropy (Log Loss):** Binary Cross-Entropy is used for binary classification problems where there are only two classes (e.g., 0 or 1).

$$\text{Binary\_CE}(y_{\text{true}}, y_{\text{pred}}) = - (y_{\text{true}} * \log(y_{\text{pred}}) + (1 - y_{\text{true}}) * \log(1 - y_{\text{pred}}))$$

- **Categorical Cross-Entropy:** Categorical Cross-Entropy is used for multi-class classification problems where there are more than two classes. It measures the dissimilarity between the true label's one-hot encoded vector (y_{true}) and the predicted probability distribution (y_{pred}).

$$\text{Categorical\_CE}(y_{\text{true}}, y_{\text{pred}}) = - \sum (y_{\text{true}} * \log(y_{\text{pred}}))$$

- **Sparse Categorical Cross-Entropy:** Sparse Categorical Cross-Entropy is similar to Categorical Cross-Entropy but is used when the true labels are provided as integers instead of one-hot encoded vectors. This can save memory in cases with a large number of classes.

These are some of the common loss functions used for different types of problems. Selecting the appropriate loss function is critical for achieving optimal performance in Neural Network training. The choice is often based on the specific problem domain and the type of data being processed.

2.4. Optimization Algorithms in Neural Networks

Optimization algorithms play a crucial role in training neural networks. They are responsible for adjusting the weights and biases of the network to minimize the loss function and make accurate predictions. Here are two commonly used optimization algorithms in neural networks:

1. Gradient Descent:

Gradient Descent is a first-order optimization algorithm that aims to find the minimum of a loss function by iteratively updating the model's parameters (weights and biases) in the opposite direction of the gradient. The gradient points in the direction of the steepest increase of the loss function, so moving in the opposite direction will lead towards the minimum.

Steps of Gradient Descent:

1. Compute the gradient of the loss function with respect to each model parameter.
2. Update the parameters by moving in the opposite direction of the gradient with a certain learning rate (step size).
3. Repeat the process for a specified number of iterations or until convergence.

Types of Gradient Descent:

- **Batch Gradient Descent:** Computes the gradient using the entire training dataset at each iteration, which can be computationally expensive for large datasets.
- **Stochastic Gradient Descent (SGD):** Computes the gradient using only one training sample at each iteration, which reduces computation but introduces more variance in the updates.
- **Mini-batch Gradient Descent:** Computes the gradient using a small batch of training samples at each iteration, striking a balance between the batch and stochastic approaches.

2. Adam (Adaptive Moment Estimation):

Adam is an adaptive learning rate optimization algorithm that combines the advantages of both RMSprop and momentum methods. It adapts the learning rates of each parameter based on the historical gradients, which helps in converging faster and avoiding getting stuck in local minima.

Steps of Adam:

1. Compute the gradient of the loss function with respect to each model parameter.
2. Calculate exponentially decaying average of past gradients (first moment) and squared gradients (second moment).
3. Combine the first and second moments to obtain the adaptive learning rates for each parameter.
4. Update the parameters by moving in the direction of the adaptive learning rate.

Adam's adaptive learning rates allow it to handle different learning rates for different parameters, making it well-suited for non-stationary data and high-dimensional spaces.

These optimization algorithms, along with their variations, are used in training neural networks to find the optimal set of parameters that minimize the loss function and improve the model's performance on the given task. The choice of the optimization algorithm can have a significant impact on the training process, and selecting the most appropriate one depends on the nature of the problem and the architecture of the neural network.

2.5. Regularization techniques

Regularization techniques are used in neural networks to prevent overfitting and improve generalization performance. Overfitting occurs when the model memorizes the training data too well and performs poorly on new, unseen data. Regularization methods introduce constraints on the model parameters during training, which helps in reducing the model's complexity and prevents it from becoming too specialized to the training data. Here are two common regularization techniques used in neural networks:

1. Dropout:

Dropout is a regularization technique introduced by Srivastava et al. in 2014. During training, dropout randomly deactivates (sets to zero) a fraction of neurons in a layer with a certain probability, often denoted as "p." This means that during each training iteration, different neurons are dropped out, effectively creating an ensemble of smaller sub-networks. During testing or inference, all neurons are active, but their weights are scaled by the dropout probability to account for the dropout during training.

Significance of Dropout:

- Reduces the co-adaptation of neurons, forcing the network to learn more robust and generalized features.
- Acts as an approximate form of model averaging, which can improve the model's performance.

2. L2 Regularization (Weight Decay):

L2 regularization, also known as weight decay, adds a penalty term to the loss function that is proportional to the square of the model's weights. The penalty term encourages the model to have smaller weights, reducing the complexity of the model and preventing overfitting.

L2 Regularization Term:

$$\text{L2\_regularization} = \lambda * \sum(w_i^2)$$

where λ is the regularization strength hyperparameter, and w_i represents each weight in the model.

Significance of L2 Regularization:

- Reduces overfitting by discouraging the model from relying heavily on any single feature.
- Encourages the model to use all features more evenly.

Both Dropout and L2 regularization are effective techniques to prevent overfitting and improve generalization in neural networks. They can be used individually or in combination to achieve better performance on various tasks. It's important to tune the hyperparameters

of these regularization techniques, such as dropout probability or the regularization strength, to find the optimal balance between preventing overfitting and preserving model performance. Additionally, other regularization techniques like L1 regularization, early stopping, and data augmentation can also be utilized to further enhance the model's robustness and generalization capabilities.

2.6. Iris Classification - Classic Example

The following table depicts an excerpt of the Iris dataset, which is a classic example in the field of machine learning. The Iris dataset contains the measurements of 150 Iris flowers from three different species—Setosa, Versicolor, and Virginica. Here, each flower example represents one row in our dataset, and the flower measurements in centimeters are stored as columns, which we also call the features of the dataset:

	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

We will follow the common convention to represent each example as a separate row in a feature matrix, X , where each feature is stored as a separate column. The Iris dataset, consisting of 150 examples and four features, can then be written as a 150×4 matrix.

We will take an object-oriented approach to defining the perceptron interface as a Python class, which will allow us to initialize new Perceptron objects that can learn from data via a 'fit' method, and make predictions via a separate predict method. As a convention, we append an underscore `_` to attributes that are not created upon the initialization of the object, but we do this by calling the object's other methods, for example, `self.w_`.

The following is the implementation of a perceptron in Python:

```
import numpy as np

class Perceptron(object):
    """Perceptron classifier.

    Parameters
```



```

-----
eta : float
    Learning rate (between 0.0 and 1.0)
n_iter : int
    Passes over the training dataset.
random_state : int
    Random number generator seed for random weight
    initialization.

Attributes
-----
w_ : 1d-array
    Weights after fitting.
errors_ : list
    Number of misclassifications (updates) in each epoch.
"""
def __init__(self, eta=0.01, n_iter=50, random_state=1):
    self.eta = eta
    self.n_iter = n_iter
    self.random_state = random_state

def fit(self, X, y):
    """Fit training data.

    Parameters
    -----
    X : {array-like}, shape = [n_examples, n_features]
        Training vectors, where n_examples is the number of
examples and
        n_features is the number of features.
    y : array-like, shape = [n_examples]
        Target values.

    Returns
    -----
    self : object
    """
    rgen = np.random.RandomState(self.random_state)
    self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 +
X.shape[1])
    self.errors_ = []

    for _ in range(self.n_iter):
        errors = 0
        for xi, target in zip(X, y):
            update = self.eta * (target - self.predict(xi))
            self.w_[1:] += update * xi
            self.w_[0] += update
            errors += int(update != 0.0)
        self.errors_.append(errors)
    return self

```

```
def net_input(self, X):
    """Calculate net input"""
    return np.dot(X, self.w_[1:]) + self.w_[0]

def predict(self, X):
    """Return class label after unit step"""
    return np.where(self.net_input(X) >= 0.0, 1, -1)
```

Using this perceptron implementation, we can now initialize new `Perceptron` objects with a given learning rate, `eta`, and the number of epochs, `n_iter` (passes over the training dataset).

Via the `fit` method, we initialize the weights in `self.w_` to a vector, $\mathbb{R}^{m+1}$, where m stands for the number of dimensions (features) in the dataset, and we add 1 for the first element in this vector that represents the bias unit. Remember that the first element in this vector, `self.w_[0]`, represents the so-called bias unit.

Also notice that this vector contains small random numbers drawn from a normal distribution with standard deviation 0.01 via `rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])`, where `rgen` is a **NumPy** random number generator that we seeded with a user-specified random seed so that we can reproduce previous results if desired.

It is important to keep in mind that we don't initialize the weights to zero because the learning rate, η (`eta`), only has an effect on the classification outcome if the weights are initialized to non-zero values. If all the weights are initialized to zero, the learning rate parameter, `eta`, affects only the scale of the weight vector, not the direction. If you are familiar with trigonometry, consider a vector, $\mathbf{v}_1 = (1, 2, 3)$, where the angle between $\mathbf{v}_1$ and a vector, $\mathbf{v}_2 = 0.5 \times \mathbf{v}_1$, would be exactly zero, as demonstrated by the following code snippet:

```
v1 = np.array([1, 2, 3])
v2 = 0.5 * v1
np.arccos(v1.dot(v2) / (np.linalg.norm(v1) * np.linalg.norm(v2)))
0.0
```

Here, `np.arccos` is the trigonometric inverse cosine, and `np.linalg.norm` is a function that computes the length of a vector (our decision to draw the random numbers from a random normal distribution — for example, instead of from a uniform distribution — and to use a standard deviation of 0.01 was arbitrary; remember, we are just interested in small random values to avoid the properties of all-zero vectors).

After the weights have been initialized, the `fit` method loops over all individual examples in the training dataset and updates the weights according to the perceptron learning rule.

The class labels are predicted by the `predict` method, which is called in the `fit` method during training to get the class label for the weight update; but `predict` can also be used to predict the class labels of new data after we have fitted our model. Furthermore, we also collect the number of misclassifications during each epoch in the `self.errors_` list so that we can later analyze how well our perceptron performed during the training. The `np.dot` function that is used in the `net_input` method simply calculates the vector dot product, $\mathbf{w}^T \mathbf{x}$.

To test our perceptron implementation, we will restrict the following analyses and examples to two feature variables (dimensions). Although the perceptron rule is not restricted to two dimensions, considering only two features, *sepal length* and *petal length*, will allow us to visualize the decision regions of the trained model in a scatter plot for learning purposes. Note that we will also only consider two flower classes, *Setosa* and *Versicolor*, from the **Iris dataset** for practical reasons—remember, the perceptron is a binary classifier. However, the perceptron algorithm can be extended to multi-class classification — for example, the one-vs.-all (*OvA*) technique.

First, we will use the **pandas** library to load the Iris dataset directly from the *UCI Machine Learning Repository* into a **DataFrame** object and print the last five lines via the **tail** method to check that the data was loaded correctly:

```
import os
import pandas as pd

s =
'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris
.data'
print('URL:', s)

df = pd.read_csv(s,
                 header=None,
                 encoding='utf-8')

df.tail()

URL:
https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.
data

{"summary":{"\n  \"name\": \"df\", \n  \"rows\": 5, \n  \"fields\": [\n    {\n      \"column\": 0, \n      \"properties\": {\n        \"dtype\": \"number\", \n        \"std\": 0.30331501776206193, \n        \"min\": 5.9, \n        \"max\": 6.7, \n        \"num_unique_values\": 5, \n        \"samples\": [\n          6.3, \n          5.9, \n          6.5\n        ], \n        \"semantic_type\": \"\", \n        \"description\": \"\"\n      }, \n      \"column\": 1, \n      \"properties\": {\n        \"dtype\": \"number\", \n        \"std\": 0.31937438845342625, \n        \"min\": 2.5, \n        \"max\": 3.4, \n        \"num_unique_values\": 3, \n        \"samples\": [\n          3.0, \n          2.5, \n          3.4\n        ], \n        \"semantic_type\": \"\", \n        \"description\": \"\"\n      }, \n      \"column\": 2, \n      \"properties\": {\n        \"dtype\": \"number\", \n        \"std\": 0.14832396974191348, \n        \"min\": 5.0, \n        \"max\": 5.4, \n        \"num_unique_values\": 4, \n        \"samples\": [\n          5.0, \n          5.1, \n          5.2\n        ], \n        \"semantic_type\": \"\", \n        \"description\": \"\"\n      }, \n      \"column\": 3, \n      \"properties\": {\n        \"dtype\": \"number\", \n        \"std\": 0.23021728866442667, \n        \"min\": 1.8, \n        \"max\": 2.3, \n        \"num_unique_values\": 4, \n        \"samples\": [\n          1.9, \n          1.8, \n          2.3\n        ]\n      }\n    ]\n  ]\n}}
```

```
],\n      \"semantic_type\": \"\", \n      \"description\": \"\"\n    },\n    {\n      \"column\": 4,\n      \"properties\":\n      {\n        \"dtype\": \"category\", \n        \"num_unique_values\":\n        1,\n        \"samples\": [\n          \"Iris-virginica\"\n        ],\n        \"semantic_type\": \"\", \n        \"description\": \"\"\n      }\n    }\n  ],\n  \"type\": \"dataframe\"}
```

Next, we extract the first 100 class labels that correspond to the 50 Iris-setosa and 50 Iris-versicolor flowers and convert the class labels into the two integer class labels, 1 (*versicolor*) and -1 (*setosa*), that we assign to a vector, *y*, where the values method of a **pandas** DataFrame yields the corresponding NumPy representation.

Similarly, we extract the first feature column (*sepal length*) and the third feature column (*petal length*) of those 100 training examples and assign them to a feature matrix, *X*, which we can visualize via a two-dimensional scatterplot:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

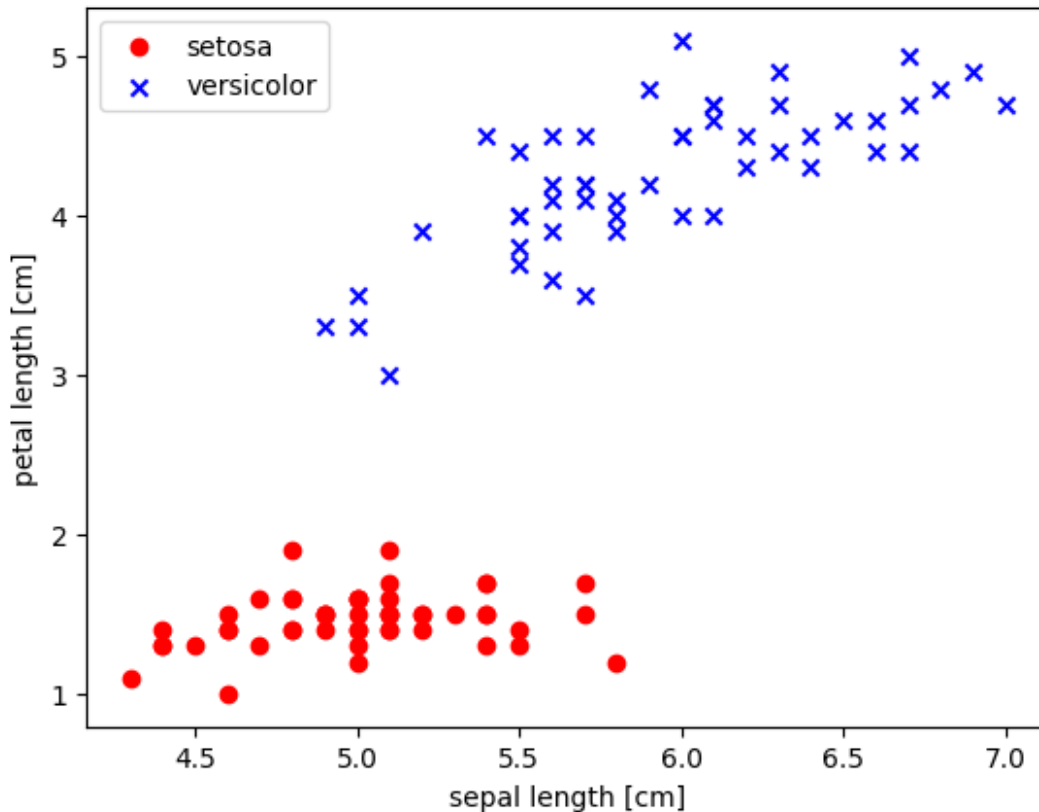
# select setosa and versicolor
y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', -1, 1)

# extract sepal length and petal length
X = df.iloc[0:100, [0, 2]].values

# plot data
plt.scatter(X[:50, 0], X[:50, 1],
            color='red', marker='o', label='setosa')
plt.scatter(X[50:100, 0], X[50:100, 1],
            color='blue', marker='x', label='versicolor')

plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.legend(loc='upper left')

# plt.savefig('images/02_06.png', dpi=300)
plt.show()
```



The preceding scatterplot shows the distribution of flower examples in the Iris dataset along the two feature axes: *petal length* and *sepal length* (measured in centimeters). In this two-dimensional feature subspace, we can see that a linear decision boundary should be sufficient to separate *Setosa* from *Versicolor* flowers.

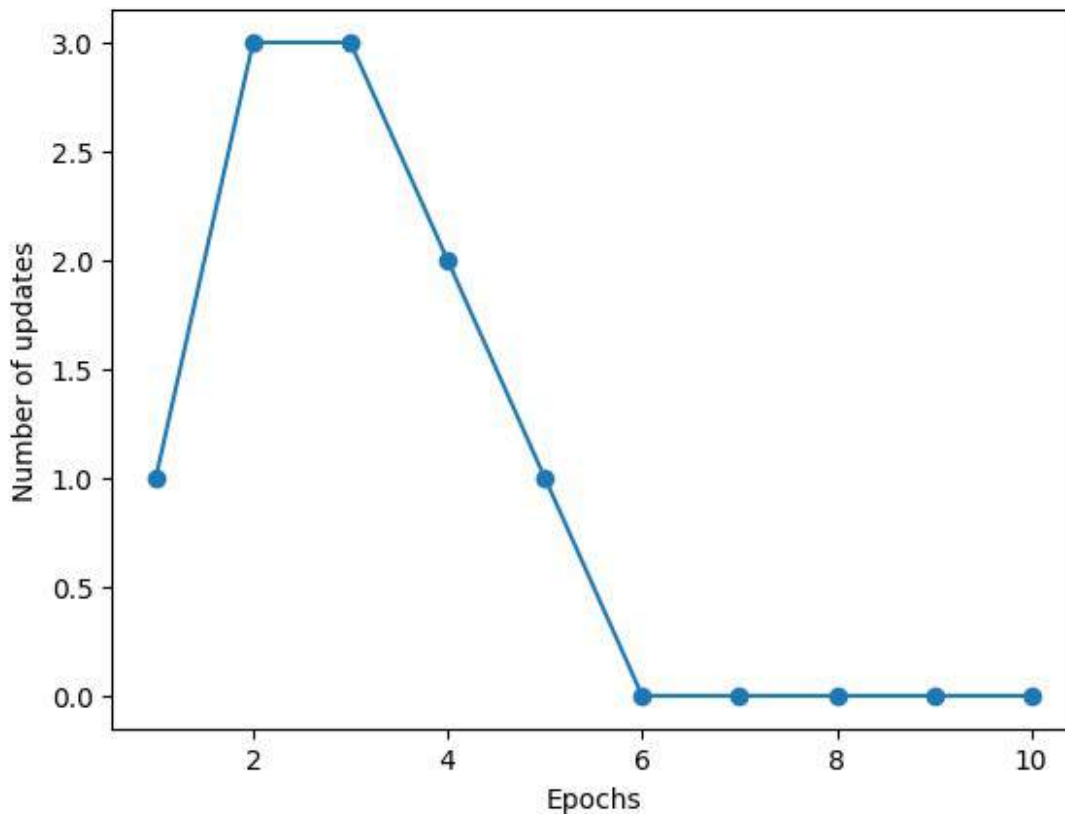
Thus, a linear classifier such as the perceptron should be able to classify the flowers in this dataset perfectly.

Now, it's time to train our perceptron algorithm on the Iris data subset that we just extracted. Also, we will plot the misclassification error for each epoch to check whether the algorithm converged and found a decision boundary that separates the two Iris flower classes:

```
ppn = Perceptron(eta=0.1, n_iter=10)
ppn.fit(X, y)

plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Number of updates')

# plt.savefig('images/02_07.png', dpi=300)
plt.show()
```



As we can see in the preceding plot, our perceptron converged after the sixth epoch and should now be able to classify the training examples perfectly. Let's implement a small convenience function to visualize the decision boundaries for two-dimensional datasets:

```
from matplotlib.colors import ListedColormap

def plot_decision_regions(X, y, classifier, resolution=0.02):

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot class examples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
```

```

alpha=0.8,
c=colors[idx],
marker=markers[idx],
label=cl,
edgecolor='black')

```

First, we define a number of colors and markers and create a colormap from the list of colors via `ListedColormap`. Then, we determine the minimum and maximum values for the two features and use those feature vectors to create a pair of grid arrays, `xx1` and `xx2`, via the NumPy `meshgrid` function. Since we trained our perceptron classifier on two feature dimensions, we need to flatten the grid arrays and create a matrix that has the same number of columns as the Iris training subset so that we can use the `predict` method to predict the class labels, `Z`, of the corresponding grid points. After reshaping the predicted class labels, `Z`, into a grid with the same dimensions as `xx1` and `xx2`, we can now draw a contour plot via Matplotlib's `contourf` function, which maps the different decision regions to different colors for each predicted class in the grid array:

```

plot_decision_regions(X, y, classifier=ppn)
plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.legend(loc='upper left')

```

```

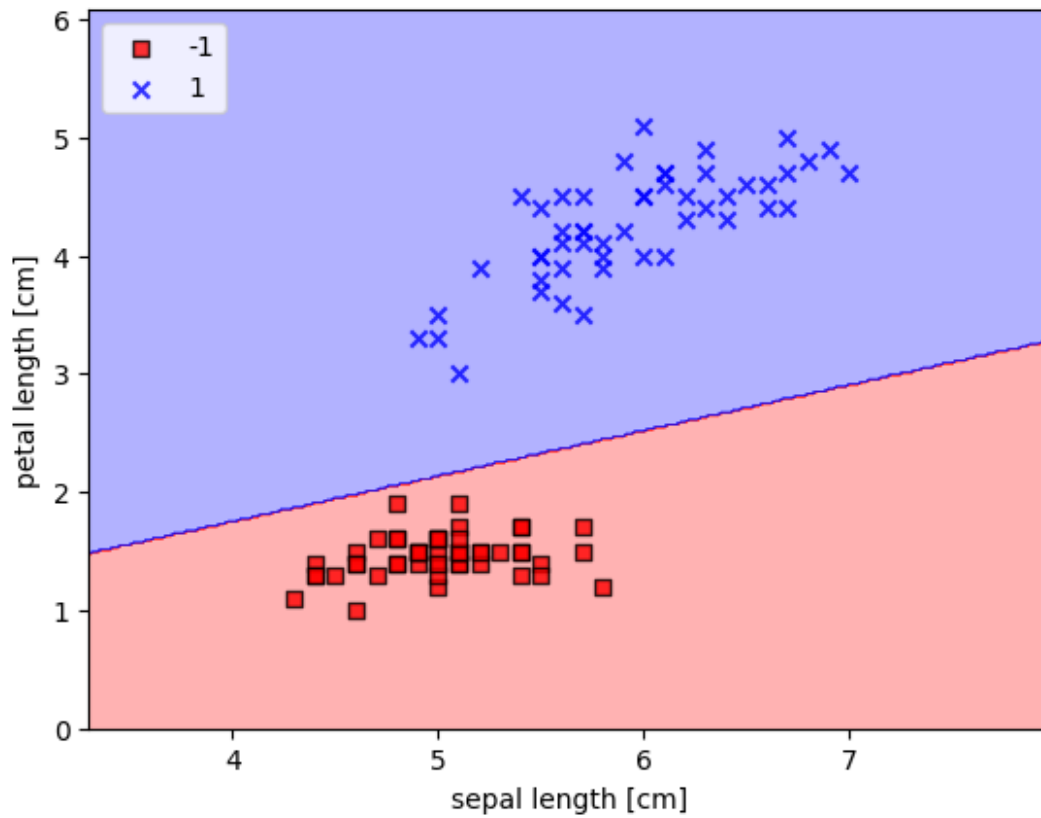
# plt.savefig('images/02_08.png', dpi=300)
plt.show()

```

```

<ipython-input-6-2dba8886e370>:24: UserWarning: You passed a
edgecolor/edgecolors ('black') for an unfilled marker ('x').
Matplotlib is ignoring the edgecolor in favor of the facecolor.
This behavior may change in the future.
  plt.scatter(x=X[y == cl, 0],

```



As we can see in the plot, the perceptron learned a decision boundary that is able to classify all flower examples in the Iris training subset perfectly.

Neural Network in AI. Part 2

1. Refresh with examples

1.1. Simple example

If a human sees a function that represents a relationship between two numbers, e.g.,

$$f(x)=3*x+1$$

it can easily figure out what will be the result if $x=10$, because we were trained in schools to solve similar tasks :).

If we want to solve this task using a computer, we can write code for a function, e.g.,

```
def function(x: float) -> float:
    return (3 * x) + 1
```

But this means that you already know the 'rules' and just pass this information to the computer.

So how would you train a neural network to do the equivalent task without telling it the exact rule to use?

We do this by using data (and a lot of it, usually)!

We must give it a set of examples with x and $f(x)$, and our neural network should be able to figure out the relationship between them without you coding it.

This is obviously a very different paradigm than what you might be used to, so let's step through it piece by piece.

Imports

Let's start with importing TensorFlow and its framework for defining a neural network as a set of Sequential layers that is called *keras*.

We also import a library called *numpy*, which helps us to represent our data as lists easily and quickly.

We also use pandas and matplotlib to visualize our training progress.

```
%%capture
%pip install tensorflow numpy matplotlib pandas

#@title Import modules
import tensorflow as tf
import numpy as np
from tensorflow import keras
from matplotlib import pyplot as plt
```

```
import pandas as pd
%matplotlib inline
```

Define the Neural Network

First, we create a simple neural network.

It has 1 layer, and that layer has 1 neuron, and the input shape to it is just 1 value.

```
model = tf.keras.Sequential([keras.layers.Dense(units=1,
input_shape=[1])])
```

Compile the Neural Network

Next we compile our Neural Network.

When we do, we have to specify 2 functions, a loss and an optimizer.

If you've seen lots of math for machine learning, here's where it's usually used, but in this case it's nicely encapsulated in functions for you.

To keep it simple, the *magic* that happens here can be described as follows:

We know that in our function, the relationship between the numbers is $f(x)=3*x+1$.

When the computer is trying to 'learn' that, it makes a guess...maybe $f(x)=10*x+10$. The **loss** function measures the guessed answers against the known correct answers and measures how well or how badly it did.

It then uses the **optimizer** function to make another guess. Based on how the **loss** function went, it will try to minimize the loss. At that point maybe it will come up with something like $f(x)=5*x+5$, which, while still pretty bad, is closer to the correct result (the loss is lower).

It will repeat this for the number of *epochs* which you will see soon. But first, here's how we tell it to use **mean squared error** for the **loss** and **stochastic gradient descent** for the **optimizer**.

You don't need to understand the math for these yet, but you can see that the *magic* works :)

```
model.compile(optimizer='sgd', loss='mean_squared_error')
```

Training Data

Next we need to have some data. For simplicity, we only have 6 combinations of x and $f(x)$. You can see that the relationship between these is that $f(x)=3*x+1$, so where $x=-1$, $f(x)=-2$, etc.

A python library called *Numpy* provides lots of array type data structures that are a defacto standard way of doing it. We declare that we want to use these by specifying the values as an `np.array`

```
x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0])
fx = np.array([-2.0, 1.0, 4.0, 7.0, 10.0, 13.0])
```

Training the Neural Network

The process of training the neural network, where it *learns* the relationship between the x and $f(x)$ is in the **model.fit** call. This is where it will go through the loop we spoke about above, making a guess, measuring how good or bad it is (the **loss**), using the **optimizer** to make another guess. It will do it for the number of epochs you specify. When you run this code, you'll see the loss on the right hand side.

```
training_history = model.fit(x, fx, epochs=100)

Epoch 1/100
1/1 [=====] - 0s 455ms/step - loss: 109.7472
Epoch 2/100
1/1 [=====] - 0s 11ms/step - loss: 86.3574
Epoch 3/100
1/1 [=====] - 0s 10ms/step - loss: 67.9552
Epoch 4/100
1/1 [=====] - 0s 15ms/step - loss: 53.4770
Epoch 5/100
1/1 [=====] - 0s 11ms/step - loss: 42.0859
Epoch 6/100
1/1 [=====] - 0s 9ms/step - loss: 33.1237
Epoch 7/100
1/1 [=====] - 0s 15ms/step - loss: 26.0724
Epoch 8/100
1/1 [=====] - 0s 11ms/step - loss: 20.5246
Epoch 9/100
1/1 [=====] - 0s 12ms/step - loss: 16.1595
Epoch 10/100
1/1 [=====] - 0s 9ms/step - loss: 12.7251
Epoch 11/100
1/1 [=====] - 0s 10ms/step - loss: 10.0228
Epoch 12/100
1/1 [=====] - 0s 8ms/step - loss: 7.8965
Epoch 13/100
1/1 [=====] - 0s 9ms/step - loss: 6.2235
Epoch 14/100
1/1 [=====] - 0s 9ms/step - loss: 4.9070
Epoch 15/100
1/1 [=====] - 0s 8ms/step - loss: 3.8710
Epoch 16/100
1/1 [=====] - 0s 9ms/step - loss: 3.0557
Epoch 17/100
1/1 [=====] - 0s 10ms/step - loss: 2.4141
Epoch 18/100
1/1 [=====] - 0s 9ms/step - loss: 1.9091
Epoch 19/100
1/1 [=====] - 0s 11ms/step - loss: 1.5115
Epoch 20/100
```

```

1/1 [=====] - 0s 15ms/step - loss: 1.1986
Epoch 21/100
1/1 [=====] - 0s 9ms/step - loss: 0.9522
Epoch 22/100
1/1 [=====] - 0s 9ms/step - loss: 0.7581
Epoch 23/100
1/1 [=====] - 0s 13ms/step - loss: 0.6053
Epoch 24/100
1/1 [=====] - 0s 8ms/step - loss: 0.4849
Epoch 25/100
1/1 [=====] - 0s 10ms/step - loss: 0.3899
Epoch 26/100
1/1 [=====] - 0s 8ms/step - loss: 0.3151
Epoch 27/100
1/1 [=====] - 0s 8ms/step - loss: 0.2560
Epoch 28/100
1/1 [=====] - 0s 9ms/step - loss: 0.2094
Epoch 29/100
1/1 [=====] - 0s 9ms/step - loss: 0.1725
Epoch 30/100
1/1 [=====] - 0s 9ms/step - loss: 0.1433
Epoch 31/100
1/1 [=====] - 0s 8ms/step - loss: 0.1202
Epoch 32/100
1/1 [=====] - 0s 11ms/step - loss: 0.1019
Epoch 33/100
1/1 [=====] - 0s 11ms/step - loss: 0.0874
Epoch 34/100
1/1 [=====] - 0s 8ms/step - loss: 0.0757
Epoch 35/100
1/1 [=====] - 0s 9ms/step - loss: 0.0665
Epoch 36/100
1/1 [=====] - 0s 8ms/step - loss: 0.0590
Epoch 37/100
1/1 [=====] - 0s 8ms/step - loss: 0.0530
Epoch 38/100
1/1 [=====] - 0s 8ms/step - loss: 0.0482
Epoch 39/100
1/1 [=====] - 0s 8ms/step - loss: 0.0442
Epoch 40/100
1/1 [=====] - 0s 12ms/step - loss: 0.0410
Epoch 41/100
1/1 [=====] - 0s 13ms/step - loss: 0.0383
Epoch 42/100
1/1 [=====] - 0s 8ms/step - loss: 0.0361
Epoch 43/100
1/1 [=====] - 0s 9ms/step - loss: 0.0342
Epoch 44/100
1/1 [=====] - 0s 11ms/step - loss: 0.0326
Epoch 45/100
1/1 [=====] - 0s 9ms/step - loss: 0.0313
Epoch 46/100
1/1 [=====] - 0s 8ms/step - loss: 0.0301

```

```
Epoch 47/100
1/1 [=====] - 0s 10ms/step - loss: 0.0290
Epoch 48/100
1/1 [=====] - 0s 9ms/step - loss: 0.0281
Epoch 49/100
1/1 [=====] - 0s 10ms/step - loss: 0.0272
Epoch 50/100
1/1 [=====] - 0s 11ms/step - loss: 0.0265
Epoch 51/100
1/1 [=====] - 0s 11ms/step - loss: 0.0258
Epoch 52/100
1/1 [=====] - 0s 8ms/step - loss: 0.0251
Epoch 53/100
1/1 [=====] - 0s 8ms/step - loss: 0.0245
Epoch 54/100
1/1 [=====] - 0s 8ms/step - loss: 0.0239
Epoch 55/100
1/1 [=====] - 0s 8ms/step - loss: 0.0233
Epoch 56/100
1/1 [=====] - 0s 9ms/step - loss: 0.0228
Epoch 57/100
1/1 [=====] - 0s 9ms/step - loss: 0.0223
Epoch 58/100
1/1 [=====] - 0s 8ms/step - loss: 0.0218
Epoch 59/100
1/1 [=====] - 0s 10ms/step - loss: 0.0213
Epoch 60/100
1/1 [=====] - 0s 9ms/step - loss: 0.0209
Epoch 61/100
1/1 [=====] - 0s 9ms/step - loss: 0.0204
Epoch 62/100
1/1 [=====] - 0s 9ms/step - loss: 0.0200
Epoch 63/100
1/1 [=====] - 0s 8ms/step - loss: 0.0196
Epoch 64/100
1/1 [=====] - 0s 8ms/step - loss: 0.0192
Epoch 65/100
1/1 [=====] - 0s 8ms/step - loss: 0.0188
Epoch 66/100
1/1 [=====] - 0s 8ms/step - loss: 0.0184
Epoch 67/100
1/1 [=====] - 0s 8ms/step - loss: 0.0180
Epoch 68/100
1/1 [=====] - 0s 10ms/step - loss: 0.0176
Epoch 69/100
1/1 [=====] - 0s 9ms/step - loss: 0.0173
Epoch 70/100
1/1 [=====] - 0s 9ms/step - loss: 0.0169
Epoch 71/100
1/1 [=====] - 0s 10ms/step - loss: 0.0166
Epoch 72/100
1/1 [=====] - 0s 9ms/step - loss: 0.0162
Epoch 73/100
```

```

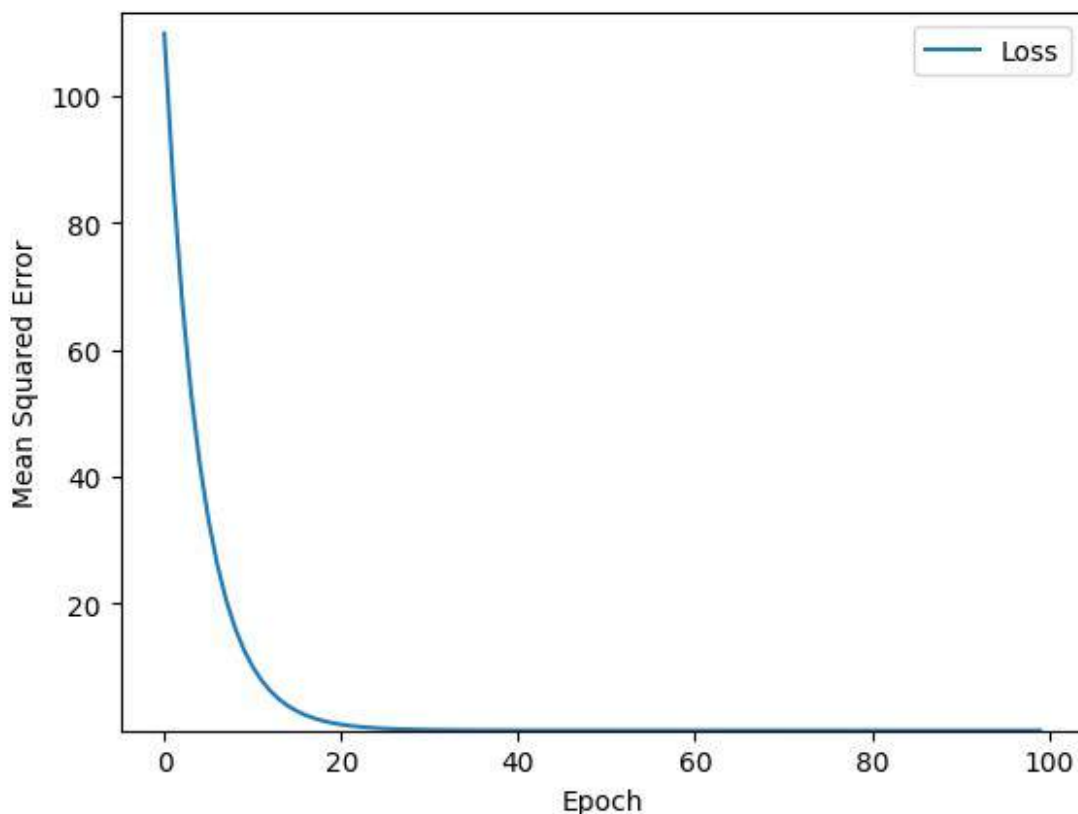
1/1 [=====] - 0s 12ms/step - loss: 0.0159
Epoch 74/100
1/1 [=====] - 0s 12ms/step - loss: 0.0156
Epoch 75/100
1/1 [=====] - 0s 14ms/step - loss: 0.0152
Epoch 76/100
1/1 [=====] - 0s 10ms/step - loss: 0.0149
Epoch 77/100
1/1 [=====] - 0s 7ms/step - loss: 0.0146
Epoch 78/100
1/1 [=====] - 0s 12ms/step - loss: 0.0143
Epoch 79/100
1/1 [=====] - 0s 12ms/step - loss: 0.0140
Epoch 80/100
1/1 [=====] - 0s 7ms/step - loss: 0.0137
Epoch 81/100
1/1 [=====] - 0s 7ms/step - loss: 0.0135
Epoch 82/100
1/1 [=====] - 0s 11ms/step - loss: 0.0132
Epoch 83/100
1/1 [=====] - 0s 7ms/step - loss: 0.0129
Epoch 84/100
1/1 [=====] - 0s 10ms/step - loss: 0.0126
Epoch 85/100
1/1 [=====] - 0s 10ms/step - loss: 0.0124
Epoch 86/100
1/1 [=====] - 0s 7ms/step - loss: 0.0121
Epoch 87/100
1/1 [=====] - 0s 14ms/step - loss: 0.0119
Epoch 88/100
1/1 [=====] - 0s 7ms/step - loss: 0.0116
Epoch 89/100
1/1 [=====] - 0s 8ms/step - loss: 0.0114
Epoch 90/100
1/1 [=====] - 0s 13ms/step - loss: 0.0112
Epoch 91/100
1/1 [=====] - 0s 8ms/step - loss: 0.0109
Epoch 92/100
1/1 [=====] - 0s 11ms/step - loss: 0.0107
Epoch 93/100
1/1 [=====] - 0s 7ms/step - loss: 0.0105
Epoch 94/100
1/1 [=====] - 0s 10ms/step - loss: 0.0103
Epoch 95/100
1/1 [=====] - 0s 7ms/step - loss: 0.0101
Epoch 96/100
1/1 [=====] - 0s 12ms/step - loss: 0.0099
Epoch 97/100
1/1 [=====] - 0s 12ms/step - loss: 0.0097
Epoch 98/100
1/1 [=====] - 0s 9ms/step - loss: 0.0095
Epoch 99/100
1/1 [=====] - 0s 9ms/step - loss: 0.0093

```

```
Epoch 100/100  
1/1 [=====] - 0s 12ms/step - loss: 0.0091
```

To have better view of the training progress, let's visualize the loss value behavior during training.

```
def plot_the_loss_curve(epochs, mse):  
    plt.figure()  
    plt.xlabel("Epoch")  
    plt.ylabel("Mean Squared Error")  
  
    plt.plot(epochs, mse, label="Loss")  
    plt.legend()  
    plt.ylim([mse.min()*0.95, mse.max() * 1.03])  
    plt.show()  
  
epochs = training_history.epoch  
hist = pd.DataFrame(training_history.history)  
rmse = hist["loss"]  
  
plot_the_loss_curve(epochs, rmse)
```



Using the Neural Network

Ok, now you have a model that has been trained to learn the relationship between x and $f(x)$. You can use the **model.predict** method to have it figure out the $f(x)$ for a previously unknown x . So, for example, if $x=10$, what do you think $f(x)$ will be? Take a guess before you run this code:

```
print(model.predict([10.0]))
1/1 [=====] - 0s 93ms/step
[[30.72188]]
```

1.2. Harder task

So we saw that one neuron is capable of predicting the value of a linear function. But can it solve more complex tasks?

Lets try the same Neural network with a parabola:

$$f(x)=x^2$$

And to help our neural network *learn* about the parabola we can increase the size of data points a bit.

```
import tensorflow as tf
import numpy as np
from tensorflow import keras
from matplotlib import pyplot as plt
import pandas as pd

%matplotlib inline

model = tf.keras.Sequential([keras.layers.Dense(units=1,
input_shape=[1])])
model.compile(optimizer='sgd', loss='mean_squared_error')
x = np.array([-4.0, -3.0, -2.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0])
fx = np.array([16.0, 9.0, 4.0, 1.0, 0.0, 1.0, 4.0, 9.0, 16.0])

training_history = model.fit(x, fx, epochs=100)

Epoch 1/100
1/1 [=====] - 0s 295ms/step - loss: 84.8880
Epoch 2/100
1/1 [=====] - 0s 11ms/step - loss: 81.5796
Epoch 3/100
1/1 [=====] - 0s 12ms/step - loss: 78.7262
Epoch 4/100
1/1 [=====] - 0s 16ms/step - loss: 76.2293
Epoch 5/100
1/1 [=====] - 0s 10ms/step - loss: 74.0141
Epoch 6/100
1/1 [=====] - 0s 16ms/step - loss: 72.0239
Epoch 7/100
1/1 [=====] - 0s 10ms/step - loss: 70.2157
Epoch 8/100
1/1 [=====] - 0s 9ms/step - loss: 68.5565
Epoch 9/100
1/1 [=====] - 0s 11ms/step - loss: 67.0213
Epoch 10/100
1/1 [=====] - 0s 10ms/step - loss: 65.5905
Epoch 11/100
```



```

1/1 [=====] - 0s 9ms/step - loss: 64.2493
Epoch 12/100
1/1 [=====] - 0s 9ms/step - loss: 62.9858
Epoch 13/100
1/1 [=====] - 0s 9ms/step - loss: 61.7908
Epoch 14/100
1/1 [=====] - 0s 9ms/step - loss: 60.6571
Epoch 15/100
1/1 [=====] - 0s 10ms/step - loss: 59.5788
Epoch 16/100
1/1 [=====] - 0s 10ms/step - loss: 58.5510
Epoch 17/100
1/1 [=====] - 0s 9ms/step - loss: 57.5698
Epoch 18/100
1/1 [=====] - 0s 9ms/step - loss: 56.6318
Epoch 19/100
1/1 [=====] - 0s 10ms/step - loss: 55.7344
Epoch 20/100
1/1 [=====] - 0s 10ms/step - loss: 54.8750
Epoch 21/100
1/1 [=====] - 0s 10ms/step - loss: 54.0514
Epoch 22/100
1/1 [=====] - 0s 11ms/step - loss: 53.2620
Epoch 23/100
1/1 [=====] - 0s 11ms/step - loss: 52.5048
Epoch 24/100
1/1 [=====] - 0s 10ms/step - loss: 51.7784
Epoch 25/100
1/1 [=====] - 0s 13ms/step - loss: 51.0814
Epoch 26/100
1/1 [=====] - 0s 10ms/step - loss: 50.4124
Epoch 27/100
1/1 [=====] - 0s 10ms/step - loss: 49.7703
Epoch 28/100
1/1 [=====] - 0s 11ms/step - loss: 49.1538
Epoch 29/100
1/1 [=====] - 0s 10ms/step - loss: 48.5619
Epoch 30/100
1/1 [=====] - 0s 11ms/step - loss: 47.9936
Epoch 31/100
1/1 [=====] - 0s 10ms/step - loss: 47.4480
Epoch 32/100
1/1 [=====] - 0s 10ms/step - loss: 46.9240
Epoch 33/100
1/1 [=====] - 0s 13ms/step - loss: 46.4208
Epoch 34/100
1/1 [=====] - 0s 11ms/step - loss: 45.9376
Epoch 35/100
1/1 [=====] - 0s 10ms/step - loss: 45.4736
Epoch 36/100
1/1 [=====] - 0s 10ms/step - loss: 45.0280
Epoch 37/100
1/1 [=====] - 0s 10ms/step - loss: 44.6000

```

```

Epoch 38/100
1/1 [=====] - 0s 12ms/step - loss: 44.1890
Epoch 39/100
1/1 [=====] - 0s 8ms/step - loss: 43.7943
Epoch 40/100
1/1 [=====] - 0s 10ms/step - loss: 43.4152
Epoch 41/100
1/1 [=====] - 0s 9ms/step - loss: 43.0511
Epoch 42/100
1/1 [=====] - 0s 8ms/step - loss: 42.7015
Epoch 43/100
1/1 [=====] - 0s 9ms/step - loss: 42.3657
Epoch 44/100
1/1 [=====] - 0s 11ms/step - loss: 42.0432
Epoch 45/100
1/1 [=====] - 0s 10ms/step - loss: 41.7335
Epoch 46/100
1/1 [=====] - 0s 9ms/step - loss: 41.4360
Epoch 47/100
1/1 [=====] - 0s 8ms/step - loss: 41.1504
Epoch 48/100
1/1 [=====] - 0s 9ms/step - loss: 40.8760
Epoch 49/100
1/1 [=====] - 0s 8ms/step - loss: 40.6125
Epoch 50/100
1/1 [=====] - 0s 9ms/step - loss: 40.3595
Epoch 51/100
1/1 [=====] - 0s 9ms/step - loss: 40.1164
Epoch 52/100
1/1 [=====] - 0s 8ms/step - loss: 39.8830
Epoch 53/100
1/1 [=====] - 0s 10ms/step - loss: 39.6588
Epoch 54/100
1/1 [=====] - 0s 12ms/step - loss: 39.4436
Epoch 55/100
1/1 [=====] - 0s 11ms/step - loss: 39.2368
Epoch 56/100
1/1 [=====] - 0s 10ms/step - loss: 39.0382
Epoch 57/100
1/1 [=====] - 0s 8ms/step - loss: 38.8475
Epoch 58/100
1/1 [=====] - 0s 10ms/step - loss: 38.6643
Epoch 59/100
1/1 [=====] - 0s 9ms/step - loss: 38.4884
Epoch 60/100
1/1 [=====] - 0s 8ms/step - loss: 38.3195
Epoch 61/100
1/1 [=====] - 0s 8ms/step - loss: 38.1572
Epoch 62/100
1/1 [=====] - 0s 8ms/step - loss: 38.0014
Epoch 63/100
1/1 [=====] - 0s 8ms/step - loss: 37.8518
Epoch 64/100

```

```

1/1 [=====] - 0s 10ms/step - loss: 37.7080
Epoch 65/100
1/1 [=====] - 0s 8ms/step - loss: 37.5700
Epoch 66/100
1/1 [=====] - 0s 8ms/step - loss: 37.4374
Epoch 67/100
1/1 [=====] - 0s 8ms/step - loss: 37.3101
Epoch 68/100
1/1 [=====] - 0s 8ms/step - loss: 37.1878
Epoch 69/100
1/1 [=====] - 0s 8ms/step - loss: 37.0704
Epoch 70/100
1/1 [=====] - 0s 8ms/step - loss: 36.9576
Epoch 71/100
1/1 [=====] - 0s 8ms/step - loss: 36.8493
Epoch 72/100
1/1 [=====] - 0s 7ms/step - loss: 36.7452
Epoch 73/100
1/1 [=====] - 0s 8ms/step - loss: 36.6453
Epoch 74/100
1/1 [=====] - 0s 7ms/step - loss: 36.5494
Epoch 75/100
1/1 [=====] - 0s 7ms/step - loss: 36.4572
Epoch 76/100
1/1 [=====] - 0s 6ms/step - loss: 36.3687
Epoch 77/100
1/1 [=====] - 0s 8ms/step - loss: 36.2837
Epoch 78/100
1/1 [=====] - 0s 8ms/step - loss: 36.2021
Epoch 79/100
1/1 [=====] - 0s 9ms/step - loss: 36.1237
Epoch 80/100
1/1 [=====] - 0s 8ms/step - loss: 36.0484
Epoch 81/100
1/1 [=====] - 0s 7ms/step - loss: 35.9761
Epoch 82/100
1/1 [=====] - 0s 9ms/step - loss: 35.9066
Epoch 83/100
1/1 [=====] - 0s 12ms/step - loss: 35.8399
Epoch 84/100
1/1 [=====] - 0s 11ms/step - loss: 35.7758
Epoch 85/100
1/1 [=====] - 0s 8ms/step - loss: 35.7143
Epoch 86/100
1/1 [=====] - 0s 8ms/step - loss: 35.6552
Epoch 87/100
1/1 [=====] - 0s 6ms/step - loss: 35.5985
Epoch 88/100
1/1 [=====] - 0s 6ms/step - loss: 35.5440
Epoch 89/100
1/1 [=====] - 0s 6ms/step - loss: 35.4916
Epoch 90/100
1/1 [=====] - 0s 6ms/step - loss: 35.4414

```

```

Epoch 91/100
1/1 [=====] - 0s 9ms/step - loss: 35.3931
Epoch 92/100
1/1 [=====] - 0s 9ms/step - loss: 35.3467
Epoch 93/100
1/1 [=====] - 0s 6ms/step - loss: 35.3022
Epoch 94/100
1/1 [=====] - 0s 6ms/step - loss: 35.2594
Epoch 95/100
1/1 [=====] - 0s 10ms/step - loss: 35.2184
Epoch 96/100
1/1 [=====] - 0s 6ms/step - loss: 35.1789
Epoch 97/100
1/1 [=====] - 0s 6ms/step - loss: 35.1410
Epoch 98/100
1/1 [=====] - 0s 9ms/step - loss: 35.1046
Epoch 99/100
1/1 [=====] - 0s 9ms/step - loss: 35.0697
Epoch 100/100
1/1 [=====] - 0s 7ms/step - loss: 35.0361

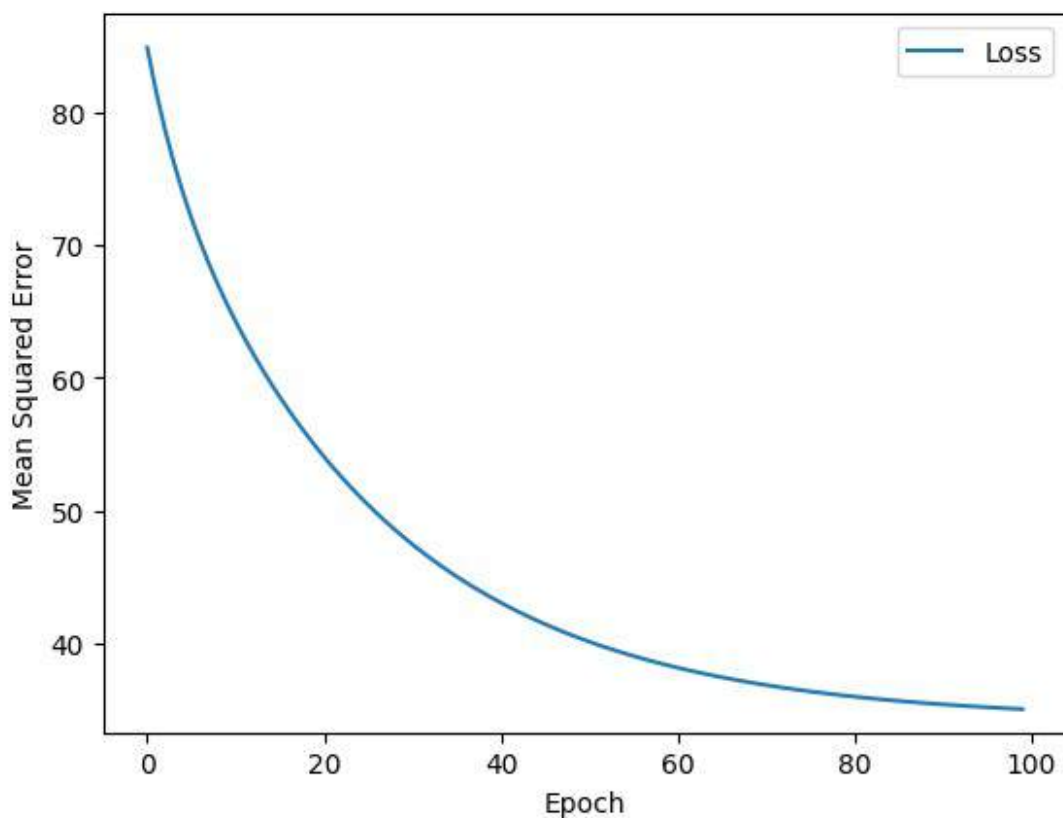
```

```

epochs = training_history.epoch
hist = pd.DataFrame(training_history.history)
rmse = hist["loss"]

plot_the_loss_curve(epochs, rmse)

```



We already see that something is wrong, as the error is very high (though the training did reduce it). Let's predict the value of $f(x)$ where $x=10$, which should be somewhere near the correct answer 100.

```
print(model.predict([10.0]))  
1/1 [=====] - 0s 70ms/step  
[[5.7825313]]
```

So what can **we** learn here? *The higher complexity of the problem, the more complex neural network we must build.* In general, it means that for real world problems your neural networks will have multiple hidden (intermediate) layers, which in turn will have multiple neurons.

Let's try to solve our problem by increasing the complexity of our neural network.

We can start by adding 1 additional neuron to our layer. So we will have a 1 value as an input, but we will have 2 values of output with our two neurons. To fix this we can add additional layer with a 1 neuron, which means we will have a single output. The complete model is as follows:

```
model = tf.keras.Sequential(  
    [  
        keras.layers.Dense(units=2, input_shape=[1]),  
        keras.layers.Dense(units=1)  
    ]  
)
```

Let's see if this helps.

```
model.compile(optimizer='sgd', loss='mean_squared_error')  
x = np.array([-4.0, -3.0, -2.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0])  
fx = np.array([16.0, 9.0, 4.0, 1.0, 0.0, 1.0, 4.0, 9.0, 16.0])  
  
training_history = model.fit(x, fx, epochs=100)  
  
Epoch 1/100  
1/1 [=====] - 0s 362ms/step - loss: 80.8950  
Epoch 2/100  
1/1 [=====] - 0s 11ms/step - loss: 77.2977  
Epoch 3/100  
1/1 [=====] - 0s 9ms/step - loss: 74.4200  
Epoch 4/100  
1/1 [=====] - 0s 9ms/step - loss: 71.8796  
Epoch 5/100  
1/1 [=====] - 0s 8ms/step - loss: 69.5034  
Epoch 6/100  
1/1 [=====] - 0s 12ms/step - loss: 67.1996  
Epoch 7/100  
1/1 [=====] - 0s 11ms/step - loss: 64.9154  
Epoch 8/100  
1/1 [=====] - 0s 9ms/step - loss: 62.6187  
Epoch 9/100
```

```

1/1 [=====] - 0s 8ms/step - loss: 60.2921
Epoch 10/100
1/1 [=====] - 0s 8ms/step - loss: 57.9299
Epoch 11/100
1/1 [=====] - 0s 10ms/step - loss: 55.5378
Epoch 12/100
1/1 [=====] - 0s 8ms/step - loss: 53.1331
Epoch 13/100
1/1 [=====] - 0s 8ms/step - loss: 50.7444
Epoch 14/100
1/1 [=====] - 0s 8ms/step - loss: 48.4098
Epoch 15/100
1/1 [=====] - 0s 8ms/step - loss: 46.1739
Epoch 16/100
1/1 [=====] - 0s 13ms/step - loss: 44.0835
Epoch 17/100
1/1 [=====] - 0s 10ms/step - loss: 42.1811
Epoch 18/100
1/1 [=====] - 0s 8ms/step - loss: 40.4999
Epoch 19/100
1/1 [=====] - 0s 10ms/step - loss: 39.0596
Epoch 20/100
1/1 [=====] - 0s 11ms/step - loss: 37.8642
Epoch 21/100
1/1 [=====] - 0s 9ms/step - loss: 36.9032
Epoch 22/100
1/1 [=====] - 0s 9ms/step - loss: 36.1540
Epoch 23/100
1/1 [=====] - 0s 9ms/step - loss: 35.5869
Epoch 24/100
1/1 [=====] - 0s 8ms/step - loss: 35.1690
Epoch 25/100
1/1 [=====] - 0s 8ms/step - loss: 34.8687
Epoch 26/100
1/1 [=====] - 0s 10ms/step - loss: 34.6575
Epoch 27/100
1/1 [=====] - 0s 11ms/step - loss: 34.5118
Epoch 28/100
1/1 [=====] - 0s 9ms/step - loss: 34.4129
Epoch 29/100
1/1 [=====] - 0s 9ms/step - loss: 34.3467
Epoch 30/100
1/1 [=====] - 0s 9ms/step - loss: 34.3029
Epoch 31/100
1/1 [=====] - 0s 13ms/step - loss: 34.2742
Epoch 32/100
1/1 [=====] - 0s 8ms/step - loss: 34.2556
Epoch 33/100
1/1 [=====] - 0s 10ms/step - loss: 34.2435
Epoch 34/100
1/1 [=====] - 0s 10ms/step - loss: 34.2358
Epoch 35/100
1/1 [=====] - 0s 10ms/step - loss: 34.2308

```

```
Epoch 36/100
1/1 [=====] - 0s 10ms/step - loss: 34.2277
Epoch 37/100
1/1 [=====] - 0s 11ms/step - loss: 34.2257
Epoch 38/100
1/1 [=====] - 0s 9ms/step - loss: 34.2244
Epoch 39/100
1/1 [=====] - 0s 9ms/step - loss: 34.2236
Epoch 40/100
1/1 [=====] - 0s 9ms/step - loss: 34.2231
Epoch 41/100
1/1 [=====] - 0s 8ms/step - loss: 34.2228
Epoch 42/100
1/1 [=====] - 0s 9ms/step - loss: 34.2226
Epoch 43/100
1/1 [=====] - 0s 8ms/step - loss: 34.2224
Epoch 44/100
1/1 [=====] - 0s 8ms/step - loss: 34.2224
Epoch 45/100
1/1 [=====] - 0s 8ms/step - loss: 34.2223
Epoch 46/100
1/1 [=====] - 0s 8ms/step - loss: 34.2223
Epoch 47/100
1/1 [=====] - 0s 7ms/step - loss: 34.2223
Epoch 48/100
1/1 [=====] - 0s 7ms/step - loss: 34.2222
Epoch 49/100
1/1 [=====] - 0s 8ms/step - loss: 34.2222
Epoch 50/100
1/1 [=====] - 0s 8ms/step - loss: 34.2222
Epoch 51/100
1/1 [=====] - 0s 7ms/step - loss: 34.2222
Epoch 52/100
1/1 [=====] - 0s 9ms/step - loss: 34.2222
Epoch 53/100
1/1 [=====] - 0s 11ms/step - loss: 34.2222
Epoch 54/100
1/1 [=====] - 0s 12ms/step - loss: 34.2222
Epoch 55/100
1/1 [=====] - 0s 9ms/step - loss: 34.2222
Epoch 56/100
1/1 [=====] - 0s 9ms/step - loss: 34.2222
Epoch 57/100
1/1 [=====] - 0s 8ms/step - loss: 34.2222
Epoch 58/100
1/1 [=====] - 0s 9ms/step - loss: 34.2222
Epoch 59/100
1/1 [=====] - 0s 9ms/step - loss: 34.2222
Epoch 60/100
1/1 [=====] - 0s 10ms/step - loss: 34.2222
Epoch 61/100
1/1 [=====] - 0s 8ms/step - loss: 34.2222
Epoch 62/100
```

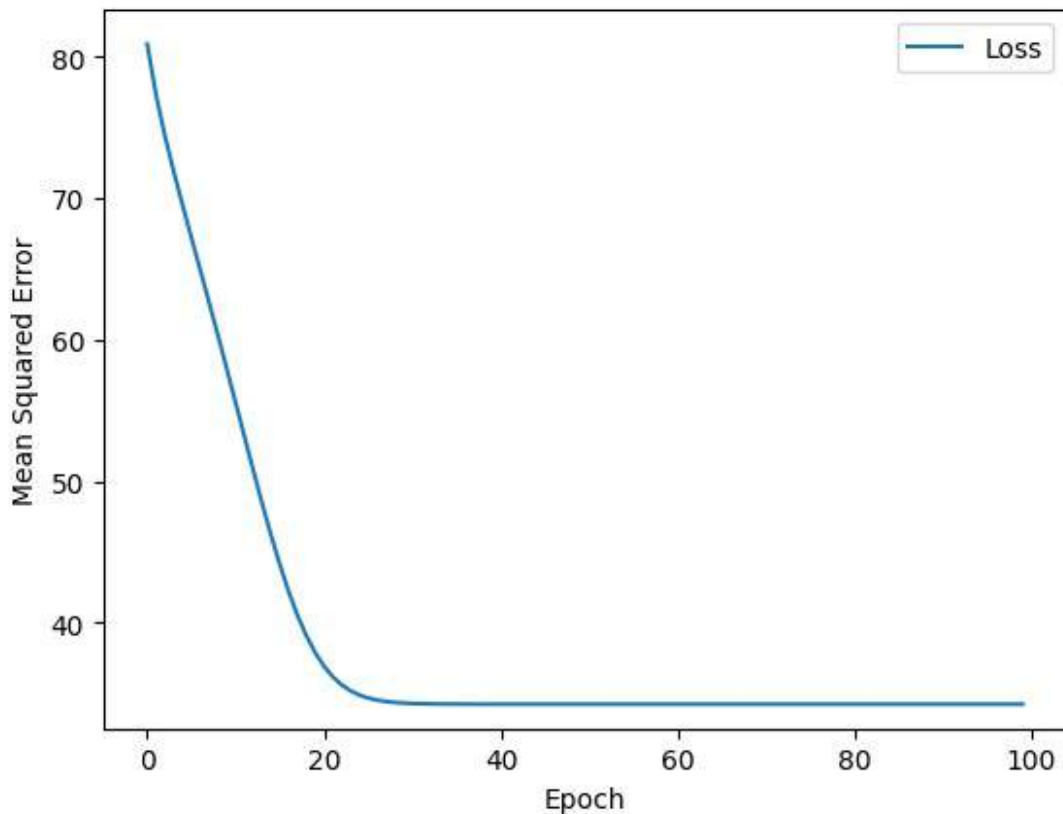
```
1/1 [=====] - 0s 8ms/step - loss: 34.2222
Epoch 63/100
1/1 [=====] - 0s 8ms/step - loss: 34.2222
Epoch 64/100
1/1 [=====] - 0s 9ms/step - loss: 34.2222
Epoch 65/100
1/1 [=====] - 0s 9ms/step - loss: 34.2222
Epoch 66/100
1/1 [=====] - 0s 8ms/step - loss: 34.2222
Epoch 67/100
1/1 [=====] - 0s 8ms/step - loss: 34.2222
Epoch 68/100
1/1 [=====] - 0s 7ms/step - loss: 34.2222
Epoch 69/100
1/1 [=====] - 0s 7ms/step - loss: 34.2222
Epoch 70/100
1/1 [=====] - 0s 8ms/step - loss: 34.2222
Epoch 71/100
1/1 [=====] - 0s 9ms/step - loss: 34.2222
Epoch 72/100
1/1 [=====] - 0s 7ms/step - loss: 34.2222
Epoch 73/100
1/1 [=====] - 0s 8ms/step - loss: 34.2222
Epoch 74/100
1/1 [=====] - 0s 10ms/step - loss: 34.2222
Epoch 75/100
1/1 [=====] - 0s 9ms/step - loss: 34.2222
Epoch 76/100
1/1 [=====] - 0s 11ms/step - loss: 34.2222
Epoch 77/100
1/1 [=====] - 0s 8ms/step - loss: 34.2222
Epoch 78/100
1/1 [=====] - 0s 8ms/step - loss: 34.2222
Epoch 79/100
1/1 [=====] - 0s 11ms/step - loss: 34.2222
Epoch 80/100
1/1 [=====] - 0s 8ms/step - loss: 34.2222
Epoch 81/100
1/1 [=====] - 0s 9ms/step - loss: 34.2222
Epoch 82/100
1/1 [=====] - 0s 10ms/step - loss: 34.2222
Epoch 83/100
1/1 [=====] - 0s 13ms/step - loss: 34.2222
Epoch 84/100
1/1 [=====] - 0s 9ms/step - loss: 34.2222
Epoch 85/100
1/1 [=====] - 0s 9ms/step - loss: 34.2222
Epoch 86/100
1/1 [=====] - 0s 10ms/step - loss: 34.2222
Epoch 87/100
1/1 [=====] - 0s 9ms/step - loss: 34.2222
Epoch 88/100
1/1 [=====] - 0s 9ms/step - loss: 34.2222
```



```
Epoch 89/100
1/1 [=====] - 0s 9ms/step - loss: 34.2222
Epoch 90/100
1/1 [=====] - 0s 9ms/step - loss: 34.2222
Epoch 91/100
1/1 [=====] - 0s 9ms/step - loss: 34.2222
Epoch 92/100
1/1 [=====] - 0s 8ms/step - loss: 34.2222
Epoch 93/100
1/1 [=====] - 0s 8ms/step - loss: 34.2222
Epoch 94/100
1/1 [=====] - 0s 8ms/step - loss: 34.2222
Epoch 95/100
1/1 [=====] - 0s 8ms/step - loss: 34.2222
Epoch 96/100
1/1 [=====] - 0s 9ms/step - loss: 34.2222
Epoch 97/100
1/1 [=====] - 0s 7ms/step - loss: 34.2222
Epoch 98/100
1/1 [=====] - 0s 9ms/step - loss: 34.2222
Epoch 99/100
1/1 [=====] - 0s 9ms/step - loss: 34.2222
Epoch 100/100
1/1 [=====] - 0s 9ms/step - loss: 34.2222

epochs = training_history.epoch
hist = pd.DataFrame(training_history.history)
rmse = hist["loss"]

plot_the_loss_curve(epochs, rmse)
```



It does look better, but still bad. So what can we do? Well first of all we can always add more neurons or layers (we will do it later).

Also, don't forget that neural networks learn from example, and having (or in our case - generating) more examples is always a good way to get better results (we'll check it out later also).

But let's try another approach here first.

When you build your neural network there are some parameters that are not tied directly to your inputs and outputs, but they influence the behaviour of your model. They are called *hyperparameters*.

Examples of hyperparameters, among others, can be an activation function (*how your neuron is triggered*) and training time.

Let's try using another activation function instead of default one.

```
model = tf.keras.Sequential(
    [
        keras.layers.Dense(units=2, activation='relu',
input_shape=[1]),
        keras.layers.Dense(units=1)
    ]
)
model.compile(optimizer='sgd', loss='mean_squared_error')
x = np.array([-4.0, -3.0, -2.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0])
fx = np.array([16.0, 9.0, 4.0, 1.0, 0.0, 1.0, 4.0, 9.0, 16.0])
```

```

training_history = model.fit(x, fx, epochs=100)

Epoch 1/100
1/1 [=====] - 0s 348ms/step - loss: 57.0397
Epoch 2/100
1/1 [=====] - 0s 9ms/step - loss: 50.4938
Epoch 3/100
1/1 [=====] - 0s 9ms/step - loss: 45.5356
Epoch 4/100
1/1 [=====] - 0s 10ms/step - loss: 42.3495
Epoch 5/100
1/1 [=====] - 0s 9ms/step - loss: 40.5857
Epoch 6/100
1/1 [=====] - 0s 11ms/step - loss: 39.6681
Epoch 7/100
1/1 [=====] - 0s 8ms/step - loss: 39.1440
Epoch 8/100
1/1 [=====] - 0s 9ms/step - loss: 38.7737
Epoch 9/100
1/1 [=====] - 0s 9ms/step - loss: 38.4587
Epoch 10/100
1/1 [=====] - 0s 9ms/step - loss: 38.1645
Epoch 11/100
1/1 [=====] - 0s 8ms/step - loss: 37.8803
Epoch 12/100
1/1 [=====] - 0s 9ms/step - loss: 37.6024
Epoch 13/100
1/1 [=====] - 0s 8ms/step - loss: 37.3298
Epoch 14/100
1/1 [=====] - 0s 8ms/step - loss: 37.0633
Epoch 15/100
1/1 [=====] - 0s 8ms/step - loss: 36.8095
Epoch 16/100
1/1 [=====] - 0s 8ms/step - loss: 36.5613
Epoch 17/100
1/1 [=====] - 0s 8ms/step - loss: 36.3189
Epoch 18/100
1/1 [=====] - 0s 9ms/step - loss: 36.0817
Epoch 19/100
1/1 [=====] - 0s 9ms/step - loss: 35.8495
Epoch 20/100
1/1 [=====] - 0s 8ms/step - loss: 35.6221
Epoch 21/100
1/1 [=====] - 0s 7ms/step - loss: 35.3996
Epoch 22/100
1/1 [=====] - 0s 7ms/step - loss: 35.1816
Epoch 23/100
1/1 [=====] - 0s 7ms/step - loss: 34.9682
Epoch 24/100
1/1 [=====] - 0s 8ms/step - loss: 34.7592
Epoch 25/100
1/1 [=====] - 0s 8ms/step - loss: 34.5545
Epoch 26/100

```

```

1/1 [=====] - 0s 9ms/step - loss: 34.3540
Epoch 27/100
1/1 [=====] - 0s 10ms/step - loss: 34.1575
Epoch 28/100
1/1 [=====] - 0s 10ms/step - loss: 33.9650
Epoch 29/100
1/1 [=====] - 0s 7ms/step - loss: 33.7764
Epoch 30/100
1/1 [=====] - 0s 8ms/step - loss: 33.5916
Epoch 31/100
1/1 [=====] - 0s 11ms/step - loss: 33.4104
Epoch 32/100
1/1 [=====] - 0s 15ms/step - loss: 33.2328
Epoch 33/100
1/1 [=====] - 0s 11ms/step - loss: 33.0587
Epoch 34/100
1/1 [=====] - 0s 12ms/step - loss: 32.8880
Epoch 35/100
1/1 [=====] - 0s 9ms/step - loss: 32.7206
Epoch 36/100
1/1 [=====] - 0s 9ms/step - loss: 32.5564
Epoch 37/100
1/1 [=====] - 0s 9ms/step - loss: 32.3953
Epoch 38/100
1/1 [=====] - 0s 10ms/step - loss: 32.2372
Epoch 39/100
1/1 [=====] - 0s 16ms/step - loss: 32.0821
Epoch 40/100
1/1 [=====] - 0s 12ms/step - loss: 31.9299
Epoch 41/100
1/1 [=====] - 0s 10ms/step - loss: 31.7805
Epoch 42/100
1/1 [=====] - 0s 10ms/step - loss: 31.6338
Epoch 43/100
1/1 [=====] - 0s 10ms/step - loss: 31.4897
Epoch 44/100
1/1 [=====] - 0s 10ms/step - loss: 31.3482
Epoch 45/100
1/1 [=====] - 0s 9ms/step - loss: 31.2092
Epoch 46/100
1/1 [=====] - 0s 9ms/step - loss: 31.0727
Epoch 47/100
1/1 [=====] - 0s 9ms/step - loss: 30.9384
Epoch 48/100
1/1 [=====] - 0s 9ms/step - loss: 30.8065
Epoch 49/100
1/1 [=====] - 0s 8ms/step - loss: 30.6768
Epoch 50/100
1/1 [=====] - 0s 9ms/step - loss: 30.5492
Epoch 51/100
1/1 [=====] - 0s 9ms/step - loss: 30.4238
Epoch 52/100
1/1 [=====] - 0s 8ms/step - loss: 30.3004
Epoch 53/100

```

```

1/1 [=====] - 0s 8ms/step - loss: 30.1789
Epoch 54/100
1/1 [=====] - 0s 8ms/step - loss: 30.0593
Epoch 55/100
1/1 [=====] - 0s 8ms/step - loss: 29.9416
Epoch 56/100
1/1 [=====] - 0s 13ms/step - loss: 29.8257
Epoch 57/100
1/1 [=====] - 0s 9ms/step - loss: 29.7115
Epoch 58/100
1/1 [=====] - 0s 9ms/step - loss: 29.5990
Epoch 59/100
1/1 [=====] - 0s 11ms/step - loss: 29.4881
Epoch 60/100
1/1 [=====] - 0s 9ms/step - loss: 29.3788
Epoch 61/100
1/1 [=====] - 0s 10ms/step - loss: 29.2710
Epoch 62/100
1/1 [=====] - 0s 11ms/step - loss: 29.1647
Epoch 63/100
1/1 [=====] - 0s 9ms/step - loss: 29.0598
Epoch 64/100
1/1 [=====] - 0s 9ms/step - loss: 28.9563
Epoch 65/100
1/1 [=====] - 0s 9ms/step - loss: 28.8541
Epoch 66/100
1/1 [=====] - 0s 9ms/step - loss: 28.7532
Epoch 67/100
1/1 [=====] - 0s 9ms/step - loss: 28.6535
Epoch 68/100
1/1 [=====] - 0s 14ms/step - loss: 28.5551
Epoch 69/100
1/1 [=====] - 0s 11ms/step - loss: 28.4577
Epoch 70/100
1/1 [=====] - 0s 9ms/step - loss: 28.3615
Epoch 71/100
1/1 [=====] - 0s 9ms/step - loss: 28.2663
Epoch 72/100
1/1 [=====] - 0s 9ms/step - loss: 28.1722
Epoch 73/100
1/1 [=====] - 0s 8ms/step - loss: 28.0791
Epoch 74/100
1/1 [=====] - 0s 9ms/step - loss: 27.9869
Epoch 75/100
1/1 [=====] - 0s 9ms/step - loss: 27.8956
Epoch 76/100
1/1 [=====] - 0s 10ms/step - loss: 27.8063
Epoch 77/100
1/1 [=====] - 0s 11ms/step - loss: 27.7186
Epoch 78/100
1/1 [=====] - 0s 12ms/step - loss: 27.6317
Epoch 79/100
1/1 [=====] - 0s 11ms/step - loss: 27.5457
Epoch 80/100

```

```

1/1 [=====] - 0s 12ms/step - loss: 27.4607
Epoch 81/100
1/1 [=====] - 0s 11ms/step - loss: 27.3764
Epoch 82/100
1/1 [=====] - 0s 8ms/step - loss: 27.2926
Epoch 83/100
1/1 [=====] - 0s 7ms/step - loss: 27.2095
Epoch 84/100
1/1 [=====] - 0s 7ms/step - loss: 27.1271
Epoch 85/100
1/1 [=====] - 0s 8ms/step - loss: 27.0454
Epoch 86/100
1/1 [=====] - 0s 9ms/step - loss: 26.9642
Epoch 87/100
1/1 [=====] - 0s 7ms/step - loss: 26.8837
Epoch 88/100
1/1 [=====] - 0s 10ms/step - loss: 26.8037
Epoch 89/100
1/1 [=====] - 0s 9ms/step - loss: 26.7242
Epoch 90/100
1/1 [=====] - 0s 9ms/step - loss: 26.6455
Epoch 91/100
1/1 [=====] - 0s 9ms/step - loss: 26.5670
Epoch 92/100
1/1 [=====] - 0s 7ms/step - loss: 26.4890
Epoch 93/100
1/1 [=====] - 0s 10ms/step - loss: 26.4115
Epoch 94/100
1/1 [=====] - 0s 6ms/step - loss: 26.3344
Epoch 95/100
1/1 [=====] - 0s 10ms/step - loss: 26.2577
Epoch 96/100
1/1 [=====] - 0s 7ms/step - loss: 26.1814
Epoch 97/100
1/1 [=====] - 0s 10ms/step - loss: 26.1055
Epoch 98/100
1/1 [=====] - 0s 9ms/step - loss: 26.0299
Epoch 99/100
1/1 [=====] - 0s 12ms/step - loss: 25.9548
Epoch 100/100
1/1 [=====] - 0s 11ms/step - loss: 25.8800

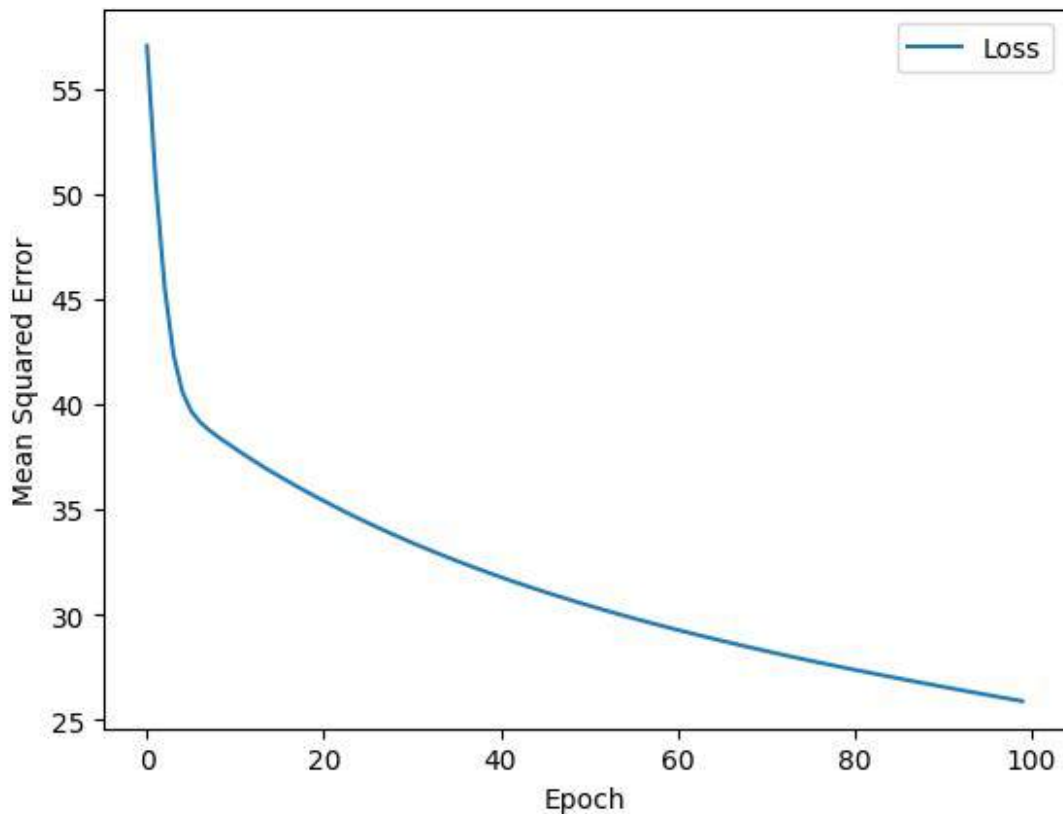
```

```

epochs = training_history.epoch
hist = pd.DataFrame(training_history.history)
rmse = hist["loss"]

plot_the_loss_curve(epochs, rmse)

```



It's better now! But let's try giving it more time.

```
model = tf.keras.Sequential(
    [
        keras.layers.Dense(units=2, activation='relu',
input_shape=[1]),
        keras.layers.Dense(units=1)
    ]
)
model.compile(optimizer='sgd', loss='mean_squared_error')
x = np.array([-4.0, -3.0, -2.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0])
fx = np.array([16.0, 9.0, 4.0, 1.0, 0.0, 1.0, 4.0, 9.0, 16.0])

training_history = model.fit(x, fx, epochs=200)
```

```
Epoch 1/200
1/1 [=====] - 0s 334ms/step - loss: 75.3329
Epoch 2/200
1/1 [=====] - 0s 9ms/step - loss: 61.5407
Epoch 3/200
1/1 [=====] - 0s 9ms/step - loss: 49.9929
Epoch 4/200
1/1 [=====] - 0s 8ms/step - loss: 39.3508
Epoch 5/200
1/1 [=====] - 0s 9ms/step - loss: 29.7058
Epoch 6/200
1/1 [=====] - 0s 8ms/step - loss: 21.5645
Epoch 7/200
1/1 [=====] - 0s 8ms/step - loss: 15.3266
```

```

Epoch 8/200
1/1 [=====] - 0s 8ms/step - loss: 11.0649
Epoch 9/200
1/1 [=====] - 0s 8ms/step - loss: 8.5086
Epoch 10/200
1/1 [=====] - 0s 8ms/step - loss: 7.1631
Epoch 11/200
1/1 [=====] - 0s 8ms/step - loss: 6.5190
Epoch 12/200
1/1 [=====] - 0s 12ms/step - loss: 6.2111
Epoch 13/200
1/1 [=====] - 0s 8ms/step - loss: 6.0420
Epoch 14/200
1/1 [=====] - 0s 10ms/step - loss: 5.9252
Epoch 15/200
1/1 [=====] - 0s 7ms/step - loss: 5.8279
Epoch 16/200
1/1 [=====] - 0s 7ms/step - loss: 5.7386
Epoch 17/200
1/1 [=====] - 0s 7ms/step - loss: 5.6537
Epoch 18/200
1/1 [=====] - 0s 7ms/step - loss: 5.5720
Epoch 19/200
1/1 [=====] - 0s 10ms/step - loss: 5.4930
Epoch 20/200
1/1 [=====] - 0s 9ms/step - loss: 5.4166
Epoch 21/200
1/1 [=====] - 0s 9ms/step - loss: 5.3427
Epoch 22/200
1/1 [=====] - 0s 9ms/step - loss: 5.2721
Epoch 23/200
1/1 [=====] - 0s 8ms/step - loss: 5.2067
Epoch 24/200
1/1 [=====] - 0s 11ms/step - loss: 5.1464
Epoch 25/200
1/1 [=====] - 0s 9ms/step - loss: 5.0868
Epoch 26/200
1/1 [=====] - 0s 9ms/step - loss: 5.0278
Epoch 27/200
1/1 [=====] - 0s 13ms/step - loss: 4.9695
Epoch 28/200
1/1 [=====] - 0s 16ms/step - loss: 4.9117
Epoch 29/200
1/1 [=====] - 0s 12ms/step - loss: 4.8546
Epoch 30/200
1/1 [=====] - 0s 13ms/step - loss: 4.7981
Epoch 31/200
1/1 [=====] - 0s 9ms/step - loss: 4.7421
Epoch 32/200
1/1 [=====] - 0s 9ms/step - loss: 4.6868
Epoch 33/200
1/1 [=====] - 0s 10ms/step - loss: 4.6321
Epoch 34/200

```



```
1/1 [=====] - 0s 10ms/step - loss: 4.5779
Epoch 35/200
1/1 [=====] - 0s 9ms/step - loss: 4.5243
Epoch 36/200
1/1 [=====] - 0s 11ms/step - loss: 4.4713
Epoch 37/200
1/1 [=====] - 0s 11ms/step - loss: 4.4189
Epoch 38/200
1/1 [=====] - 0s 9ms/step - loss: 4.3671
Epoch 39/200
1/1 [=====] - 0s 8ms/step - loss: 4.3158
Epoch 40/200
1/1 [=====] - 0s 8ms/step - loss: 4.2651
Epoch 41/200
1/1 [=====] - 0s 8ms/step - loss: 4.2149
Epoch 42/200
1/1 [=====] - 0s 8ms/step - loss: 4.1653
Epoch 43/200
1/1 [=====] - 0s 8ms/step - loss: 4.1163
Epoch 44/200
1/1 [=====] - 0s 9ms/step - loss: 4.0678
Epoch 45/200
1/1 [=====] - 0s 10ms/step - loss: 4.0198
Epoch 46/200
1/1 [=====] - 0s 11ms/step - loss: 3.9724
Epoch 47/200
1/1 [=====] - 0s 8ms/step - loss: 3.9256
Epoch 48/200
1/1 [=====] - 0s 12ms/step - loss: 3.8793
Epoch 49/200
1/1 [=====] - 0s 7ms/step - loss: 3.8335
Epoch 50/200
1/1 [=====] - 0s 9ms/step - loss: 3.7882
Epoch 51/200
1/1 [=====] - 0s 8ms/step - loss: 3.7435
Epoch 52/200
1/1 [=====] - 0s 8ms/step - loss: 3.6993
Epoch 53/200
1/1 [=====] - 0s 8ms/step - loss: 3.6556
Epoch 54/200
1/1 [=====] - 0s 7ms/step - loss: 3.6125
Epoch 55/200
1/1 [=====] - 0s 9ms/step - loss: 3.5698
Epoch 56/200
1/1 [=====] - 0s 9ms/step - loss: 3.5277
Epoch 57/200
1/1 [=====] - 0s 8ms/step - loss: 3.4861
Epoch 58/200
1/1 [=====] - 0s 9ms/step - loss: 3.4449
Epoch 59/200
1/1 [=====] - 0s 11ms/step - loss: 3.4043
Epoch 60/200
1/1 [=====] - 0s 11ms/step - loss: 3.3642
```

```
Epoch 61/200
1/1 [=====] - 0s 8ms/step - loss: 3.3245
Epoch 62/200
1/1 [=====] - 0s 8ms/step - loss: 3.2854
Epoch 63/200
1/1 [=====] - 0s 8ms/step - loss: 3.2468
Epoch 64/200
1/1 [=====] - 0s 9ms/step - loss: 3.2086
Epoch 65/200
1/1 [=====] - 0s 9ms/step - loss: 3.1709
Epoch 66/200
1/1 [=====] - 0s 8ms/step - loss: 3.1337
Epoch 67/200
1/1 [=====] - 0s 9ms/step - loss: 3.0970
Epoch 68/200
1/1 [=====] - 0s 9ms/step - loss: 3.0607
Epoch 69/200
1/1 [=====] - 0s 9ms/step - loss: 3.0250
Epoch 70/200
1/1 [=====] - 0s 9ms/step - loss: 2.9896
Epoch 71/200
1/1 [=====] - 0s 9ms/step - loss: 2.9548
Epoch 72/200
1/1 [=====] - 0s 9ms/step - loss: 2.9204
Epoch 73/200
1/1 [=====] - 0s 10ms/step - loss: 2.8865
Epoch 74/200
1/1 [=====] - 0s 9ms/step - loss: 2.8530
Epoch 75/200
1/1 [=====] - 0s 12ms/step - loss: 2.8200
Epoch 76/200
1/1 [=====] - 0s 10ms/step - loss: 2.7874
Epoch 77/200
1/1 [=====] - 0s 9ms/step - loss: 2.7552
Epoch 78/200
1/1 [=====] - 0s 9ms/step - loss: 2.7235
Epoch 79/200
1/1 [=====] - 0s 9ms/step - loss: 2.6923
Epoch 80/200
1/1 [=====] - 0s 11ms/step - loss: 2.6614
Epoch 81/200
1/1 [=====] - 0s 13ms/step - loss: 2.6310
Epoch 82/200
1/1 [=====] - 0s 12ms/step - loss: 2.6011
Epoch 83/200
1/1 [=====] - 0s 12ms/step - loss: 2.5715
Epoch 84/200
1/1 [=====] - 0s 10ms/step - loss: 2.5424
Epoch 85/200
1/1 [=====] - 0s 10ms/step - loss: 2.5137
Epoch 86/200
1/1 [=====] - 0s 13ms/step - loss: 2.4854
Epoch 87/200
```

```

1/1 [=====] - 0s 9ms/step - loss: 2.4575
Epoch 88/200
1/1 [=====] - 0s 9ms/step - loss: 2.4301
Epoch 89/200
1/1 [=====] - 0s 11ms/step - loss: 2.4030
Epoch 90/200
1/1 [=====] - 0s 10ms/step - loss: 2.3763
Epoch 91/200
1/1 [=====] - 0s 11ms/step - loss: 2.3501
Epoch 92/200
1/1 [=====] - 0s 10ms/step - loss: 2.3242
Epoch 93/200
1/1 [=====] - 0s 9ms/step - loss: 2.2987
Epoch 94/200
1/1 [=====] - 0s 9ms/step - loss: 2.2736
Epoch 95/200
1/1 [=====] - 0s 9ms/step - loss: 2.2489
Epoch 96/200
1/1 [=====] - 0s 7ms/step - loss: 2.2245
Epoch 97/200
1/1 [=====] - 0s 12ms/step - loss: 2.2006
Epoch 98/200
1/1 [=====] - 0s 16ms/step - loss: 2.1770
Epoch 99/200
1/1 [=====] - 0s 7ms/step - loss: 2.1538
Epoch 100/200
1/1 [=====] - 0s 15ms/step - loss: 2.1309
Epoch 101/200
1/1 [=====] - 0s 10ms/step - loss: 2.1084
Epoch 102/200
1/1 [=====] - 0s 7ms/step - loss: 2.0863
Epoch 103/200
1/1 [=====] - 0s 7ms/step - loss: 2.0645
Epoch 104/200
1/1 [=====] - 0s 9ms/step - loss: 2.0431
Epoch 105/200
1/1 [=====] - 0s 7ms/step - loss: 2.0220
Epoch 106/200
1/1 [=====] - 0s 13ms/step - loss: 2.0013
Epoch 107/200
1/1 [=====] - 0s 9ms/step - loss: 1.9809
Epoch 108/200
1/1 [=====] - 0s 9ms/step - loss: 1.9609
Epoch 109/200
1/1 [=====] - 0s 11ms/step - loss: 1.9412
Epoch 110/200
1/1 [=====] - 0s 9ms/step - loss: 1.9218
Epoch 111/200
1/1 [=====] - 0s 10ms/step - loss: 1.9027
Epoch 112/200
1/1 [=====] - 0s 11ms/step - loss: 1.8840
Epoch 113/200
1/1 [=====] - 0s 13ms/step - loss: 1.8656

```

```
Epoch 114/200
1/1 [=====] - 0s 11ms/step - loss: 1.8474
Epoch 115/200
1/1 [=====] - 0s 10ms/step - loss: 1.8297
Epoch 116/200
1/1 [=====] - 0s 9ms/step - loss: 1.8122
Epoch 117/200
1/1 [=====] - 0s 10ms/step - loss: 1.7950
Epoch 118/200
1/1 [=====] - 0s 8ms/step - loss: 1.7781
Epoch 119/200
1/1 [=====] - 0s 7ms/step - loss: 1.7615
Epoch 120/200
1/1 [=====] - 0s 9ms/step - loss: 1.7452
Epoch 121/200
1/1 [=====] - 0s 14ms/step - loss: 1.7292
Epoch 122/200
1/1 [=====] - 0s 13ms/step - loss: 1.7135
Epoch 123/200
1/1 [=====] - 0s 10ms/step - loss: 1.6981
Epoch 124/200
1/1 [=====] - 0s 9ms/step - loss: 1.6830
Epoch 125/200
1/1 [=====] - 0s 8ms/step - loss: 1.6681
Epoch 126/200
1/1 [=====] - 0s 9ms/step - loss: 1.6535
Epoch 127/200
1/1 [=====] - 0s 9ms/step - loss: 1.6392
Epoch 128/200
1/1 [=====] - 0s 9ms/step - loss: 1.6251
Epoch 129/200
1/1 [=====] - 0s 9ms/step - loss: 1.6113
Epoch 130/200
1/1 [=====] - 0s 7ms/step - loss: 1.5977
Epoch 131/200
1/1 [=====] - 0s 7ms/step - loss: 1.5844
Epoch 132/200
1/1 [=====] - 0s 9ms/step - loss: 1.5714
Epoch 133/200
1/1 [=====] - 0s 7ms/step - loss: 1.5586
Epoch 134/200
1/1 [=====] - 0s 10ms/step - loss: 1.5460
Epoch 135/200
1/1 [=====] - 0s 7ms/step - loss: 1.5337
Epoch 136/200
1/1 [=====] - 0s 11ms/step - loss: 1.5217
Epoch 137/200
1/1 [=====] - 0s 8ms/step - loss: 1.5098
Epoch 138/200
1/1 [=====] - 0s 11ms/step - loss: 1.4982
Epoch 139/200
1/1 [=====] - 0s 7ms/step - loss: 1.4868
Epoch 140/200
```

```

1/1 [=====] - 0s 10ms/step - loss: 1.4756
Epoch 141/200
1/1 [=====] - 0s 9ms/step - loss: 1.4647
Epoch 142/200
1/1 [=====] - 0s 6ms/step - loss: 1.4540
Epoch 143/200
1/1 [=====] - 0s 9ms/step - loss: 1.4435
Epoch 144/200
1/1 [=====] - 0s 6ms/step - loss: 1.4331
Epoch 145/200
1/1 [=====] - 0s 11ms/step - loss: 1.4230
Epoch 146/200
1/1 [=====] - 0s 7ms/step - loss: 1.4131
Epoch 147/200
1/1 [=====] - 0s 11ms/step - loss: 1.4034
Epoch 148/200
1/1 [=====] - 0s 8ms/step - loss: 1.3939
Epoch 149/200
1/1 [=====] - 0s 9ms/step - loss: 1.3846
Epoch 150/200
1/1 [=====] - 0s 8ms/step - loss: 1.3755
Epoch 151/200
1/1 [=====] - 0s 8ms/step - loss: 1.3666
Epoch 152/200
1/1 [=====] - 0s 8ms/step - loss: 1.3578
Epoch 153/200
1/1 [=====] - 0s 12ms/step - loss: 1.3493
Epoch 154/200
1/1 [=====] - 0s 8ms/step - loss: 1.3409
Epoch 155/200
1/1 [=====] - 0s 8ms/step - loss: 1.3326
Epoch 156/200
1/1 [=====] - 0s 13ms/step - loss: 1.3246
Epoch 157/200
1/1 [=====] - 0s 8ms/step - loss: 1.3167
Epoch 158/200
1/1 [=====] - 0s 8ms/step - loss: 1.3090
Epoch 159/200
1/1 [=====] - 0s 9ms/step - loss: 1.3015
Epoch 160/200
1/1 [=====] - 0s 8ms/step - loss: 1.2941
Epoch 161/200
1/1 [=====] - 0s 9ms/step - loss: 1.2868
Epoch 162/200
1/1 [=====] - 0s 8ms/step - loss: 1.2797
Epoch 163/200
1/1 [=====] - 0s 8ms/step - loss: 1.2728
Epoch 164/200
1/1 [=====] - 0s 9ms/step - loss: 1.2660
Epoch 165/200
1/1 [=====] - 0s 9ms/step - loss: 1.2594
Epoch 166/200
1/1 [=====] - 0s 9ms/step - loss: 1.2529

```

```
Epoch 167/200
1/1 [=====] - 0s 6ms/step - loss: 1.2465
Epoch 168/200
1/1 [=====] - 0s 10ms/step - loss: 1.2403
Epoch 169/200
1/1 [=====] - 0s 7ms/step - loss: 1.2342
Epoch 170/200
1/1 [=====] - 0s 10ms/step - loss: 1.2283
Epoch 171/200
1/1 [=====] - 0s 6ms/step - loss: 1.2225
Epoch 172/200
1/1 [=====] - 0s 12ms/step - loss: 1.2168
Epoch 173/200
1/1 [=====] - 0s 11ms/step - loss: 1.2112
Epoch 174/200
1/1 [=====] - 0s 10ms/step - loss: 1.2058
Epoch 175/200
1/1 [=====] - 0s 10ms/step - loss: 1.2004
Epoch 176/200
1/1 [=====] - 0s 9ms/step - loss: 1.1952
Epoch 177/200
1/1 [=====] - 0s 8ms/step - loss: 1.1901
Epoch 178/200
1/1 [=====] - 0s 9ms/step - loss: 1.1851
Epoch 179/200
1/1 [=====] - 0s 10ms/step - loss: 1.1803
Epoch 180/200
1/1 [=====] - 0s 9ms/step - loss: 1.1755
Epoch 181/200
1/1 [=====] - 0s 11ms/step - loss: 1.1709
Epoch 182/200
1/1 [=====] - 0s 9ms/step - loss: 1.1663
Epoch 183/200
1/1 [=====] - 0s 8ms/step - loss: 1.1619
Epoch 184/200
1/1 [=====] - 0s 12ms/step - loss: 1.1575
Epoch 185/200
1/1 [=====] - 0s 8ms/step - loss: 1.1532
Epoch 186/200
1/1 [=====] - 0s 9ms/step - loss: 1.1491
Epoch 187/200
1/1 [=====] - 0s 9ms/step - loss: 1.1450
Epoch 188/200
1/1 [=====] - 0s 10ms/step - loss: 1.1410
Epoch 189/200
1/1 [=====] - 0s 9ms/step - loss: 1.1372
Epoch 190/200
1/1 [=====] - 0s 7ms/step - loss: 1.1334
Epoch 191/200
1/1 [=====] - 0s 10ms/step - loss: 1.1296
Epoch 192/200
1/1 [=====] - 0s 8ms/step - loss: 1.1260
Epoch 193/200
```

```

1/1 [=====] - 0s 8ms/step - loss: 1.1225
Epoch 194/200
1/1 [=====] - 0s 8ms/step - loss: 1.1190
Epoch 195/200
1/1 [=====] - 0s 8ms/step - loss: 1.1156
Epoch 196/200
1/1 [=====] - 0s 10ms/step - loss: 1.1123
Epoch 197/200
1/1 [=====] - 0s 14ms/step - loss: 1.1091
Epoch 198/200
1/1 [=====] - 0s 21ms/step - loss: 1.1059
Epoch 199/200
1/1 [=====] - 0s 14ms/step - loss: 1.1028
Epoch 200/200
1/1 [=====] - 0s 7ms/step - loss: 1.0998

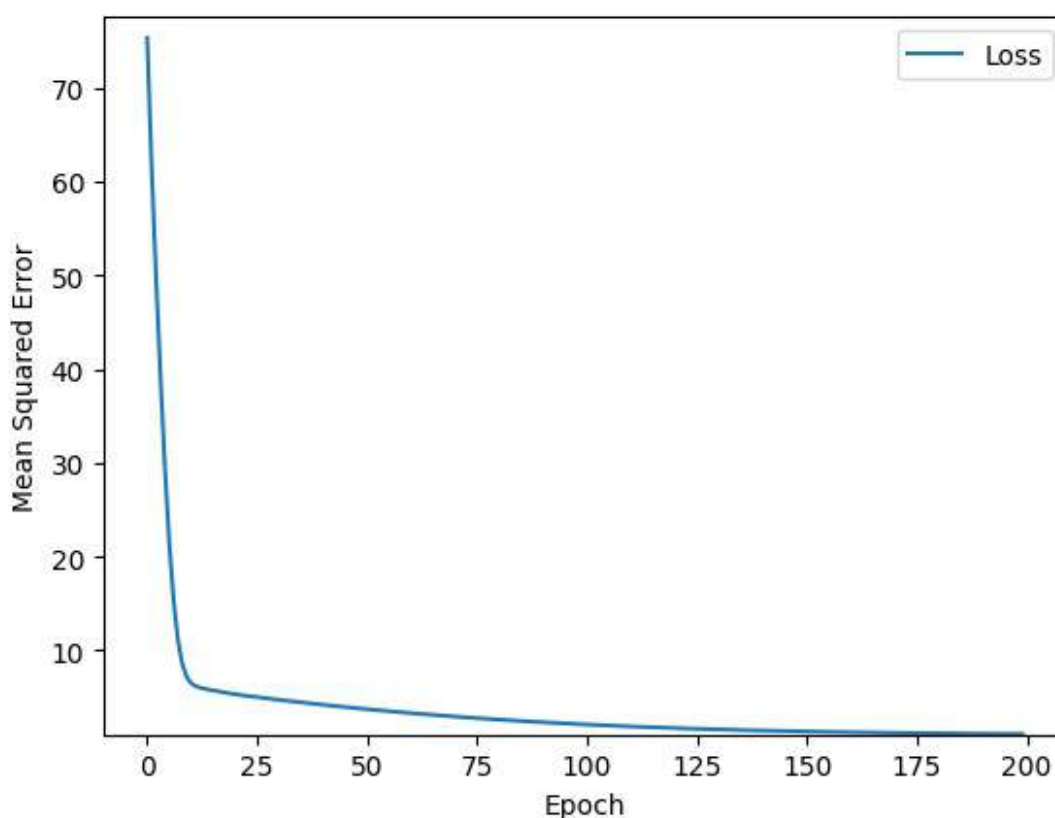
```

```

epochs = training_history.epoch
hist = pd.DataFrame(training_history.history)
rmse = hist["loss"]

```

```
plot_the_loss_curve(epochs, rmse)
```



So as you can see, changing some *hyperparameters* can positively (or negatively) influence your model. The approach to figure out the best parameters is called *hyperparameters tuning*.

Let's try adding more neurons and another layer.

```

model = tf.keras.Sequential(
    [
        keras.layers.Dense(units=8, activation='relu',
input_shape=[1]),
        keras.layers.Dense(units=8, activation='relu'),
        keras.layers.Dense(units=1)
    ]
)
model.compile(optimizer='sgd', loss='mean_squared_error')
x = np.array([-4.0, -3.0, -2.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0])
fx = np.array([16.0, 9.0, 4.0, 1.0, 0.0, 1.0, 4.0, 9.0, 16.0])

training_history = model.fit(x, fx, epochs=200)

Epoch 1/200
1/1 [=====] - 0s 399ms/step - loss: 81.3954
Epoch 2/200
1/1 [=====] - 0s 8ms/step - loss: 55.2921
Epoch 3/200
1/1 [=====] - 0s 9ms/step - loss: 42.1996
Epoch 4/200
1/1 [=====] - 0s 9ms/step - loss: 37.1924
Epoch 5/200
1/1 [=====] - 0s 11ms/step - loss: 35.6061
Epoch 6/200
1/1 [=====] - 0s 8ms/step - loss: 33.6227
Epoch 7/200
1/1 [=====] - 0s 8ms/step - loss: 30.8637
Epoch 8/200
1/1 [=====] - 0s 10ms/step - loss: 26.9811
Epoch 9/200
1/1 [=====] - 0s 9ms/step - loss: 21.8025
Epoch 10/200
1/1 [=====] - 0s 9ms/step - loss: 15.9018
Epoch 11/200
1/1 [=====] - 0s 8ms/step - loss: 11.0418
Epoch 12/200
1/1 [=====] - 0s 9ms/step - loss: 8.7025
Epoch 13/200
1/1 [=====] - 0s 8ms/step - loss: 7.9875
Epoch 14/200
1/1 [=====] - 0s 8ms/step - loss: 7.6043
Epoch 15/200
1/1 [=====] - 0s 9ms/step - loss: 7.2554
Epoch 16/200
1/1 [=====] - 0s 8ms/step - loss: 6.9273
Epoch 17/200
1/1 [=====] - 0s 8ms/step - loss: 6.6166
Epoch 18/200
1/1 [=====] - 0s 10ms/step - loss: 6.3222
Epoch 19/200
1/1 [=====] - 0s 9ms/step - loss: 6.0431
Epoch 20/200
1/1 [=====] - 0s 10ms/step - loss: 5.7787

```



```

Epoch 21/200
1/1 [=====] - 0s 8ms/step - loss: 5.5284
Epoch 22/200
1/1 [=====] - 0s 8ms/step - loss: 5.2919
Epoch 23/200
1/1 [=====] - 0s 9ms/step - loss: 5.0842
Epoch 24/200
1/1 [=====] - 0s 9ms/step - loss: 4.8844
Epoch 25/200
1/1 [=====] - 0s 9ms/step - loss: 4.6921
Epoch 26/200
1/1 [=====] - 0s 8ms/step - loss: 4.5073
Epoch 27/200
1/1 [=====] - 0s 10ms/step - loss: 4.3298
Epoch 28/200
1/1 [=====] - 0s 10ms/step - loss: 4.1592
Epoch 29/200
1/1 [=====] - 0s 9ms/step - loss: 3.9954
Epoch 30/200
1/1 [=====] - 0s 8ms/step - loss: 3.8384
Epoch 31/200
1/1 [=====] - 0s 8ms/step - loss: 3.6878
Epoch 32/200
1/1 [=====] - 0s 8ms/step - loss: 3.5440
Epoch 33/200
1/1 [=====] - 0s 9ms/step - loss: 3.4071
Epoch 34/200
1/1 [=====] - 0s 9ms/step - loss: 3.2754
Epoch 35/200
1/1 [=====] - 0s 9ms/step - loss: 3.1487
Epoch 36/200
1/1 [=====] - 0s 9ms/step - loss: 3.0264
Epoch 37/200
1/1 [=====] - 0s 9ms/step - loss: 2.9046
Epoch 38/200
1/1 [=====] - 0s 9ms/step - loss: 2.7880
Epoch 39/200
1/1 [=====] - 0s 9ms/step - loss: 2.6766
Epoch 40/200
1/1 [=====] - 0s 9ms/step - loss: 2.5702
Epoch 41/200
1/1 [=====] - 0s 13ms/step - loss: 2.4688
Epoch 42/200
1/1 [=====] - 0s 8ms/step - loss: 2.3722
Epoch 43/200
1/1 [=====] - 0s 9ms/step - loss: 2.2804
Epoch 44/200
1/1 [=====] - 0s 9ms/step - loss: 2.1932
Epoch 45/200
1/1 [=====] - 0s 9ms/step - loss: 2.1105
Epoch 46/200
1/1 [=====] - 0s 8ms/step - loss: 2.0322
Epoch 47/200

```

```
1/1 [=====] - 0s 8ms/step - loss: 1.9581
Epoch 48/200
1/1 [=====] - 0s 8ms/step - loss: 1.8881
Epoch 49/200
1/1 [=====] - 0s 9ms/step - loss: 1.8222
Epoch 50/200
1/1 [=====] - 0s 8ms/step - loss: 1.7602
Epoch 51/200
1/1 [=====] - 0s 8ms/step - loss: 1.7018
Epoch 52/200
1/1 [=====] - 0s 8ms/step - loss: 1.6469
Epoch 53/200
1/1 [=====] - 0s 8ms/step - loss: 1.5953
Epoch 54/200
1/1 [=====] - 0s 8ms/step - loss: 1.5468
Epoch 55/200
1/1 [=====] - 0s 8ms/step - loss: 1.5000
Epoch 56/200
1/1 [=====] - 0s 8ms/step - loss: 1.4491
Epoch 57/200
1/1 [=====] - 0s 8ms/step - loss: 1.4020
Epoch 58/200
1/1 [=====] - 0s 8ms/step - loss: 1.3584
Epoch 59/200
1/1 [=====] - 0s 8ms/step - loss: 1.3180
Epoch 60/200
1/1 [=====] - 0s 8ms/step - loss: 1.2798
Epoch 61/200
1/1 [=====] - 0s 8ms/step - loss: 1.2330
Epoch 62/200
1/1 [=====] - 0s 8ms/step - loss: 1.1975
Epoch 63/200
1/1 [=====] - 0s 8ms/step - loss: 1.1721
Epoch 64/200
1/1 [=====] - 0s 9ms/step - loss: 1.1487
Epoch 65/200
1/1 [=====] - 0s 8ms/step - loss: 1.1270
Epoch 66/200
1/1 [=====] - 0s 8ms/step - loss: 1.1068
Epoch 67/200
1/1 [=====] - 0s 8ms/step - loss: 1.0882
Epoch 68/200
1/1 [=====] - 0s 8ms/step - loss: 1.0709
Epoch 69/200
1/1 [=====] - 0s 8ms/step - loss: 1.0549
Epoch 70/200
1/1 [=====] - 0s 9ms/step - loss: 1.0400
Epoch 71/200
1/1 [=====] - 0s 8ms/step - loss: 1.0263
Epoch 72/200
1/1 [=====] - 0s 8ms/step - loss: 1.0136
Epoch 73/200
1/1 [=====] - 0s 8ms/step - loss: 1.0018
```

```
Epoch 74/200
1/1 [=====] - 0s 9ms/step - loss: 0.9914
Epoch 75/200
1/1 [=====] - 0s 10ms/step - loss: 0.9818
Epoch 76/200
1/1 [=====] - 0s 8ms/step - loss: 0.9721
Epoch 77/200
1/1 [=====] - 0s 11ms/step - loss: 0.9634
Epoch 78/200
1/1 [=====] - 0s 8ms/step - loss: 0.9553
Epoch 79/200
1/1 [=====] - 0s 10ms/step - loss: 0.9477
Epoch 80/200
1/1 [=====] - 0s 9ms/step - loss: 0.9407
Epoch 81/200
1/1 [=====] - 0s 10ms/step - loss: 0.9342
Epoch 82/200
1/1 [=====] - 0s 8ms/step - loss: 0.9281
Epoch 83/200
1/1 [=====] - 0s 10ms/step - loss: 0.9225
Epoch 84/200
1/1 [=====] - 0s 10ms/step - loss: 0.9172
Epoch 85/200
1/1 [=====] - 0s 19ms/step - loss: 0.9122
Epoch 86/200
1/1 [=====] - 0s 12ms/step - loss: 0.9076
Epoch 87/200
1/1 [=====] - 0s 9ms/step - loss: 0.9033
Epoch 88/200
1/1 [=====] - 0s 9ms/step - loss: 0.8992
Epoch 89/200
1/1 [=====] - 0s 10ms/step - loss: 0.8947
Epoch 90/200
1/1 [=====] - 0s 10ms/step - loss: 0.8879
Epoch 91/200
1/1 [=====] - 0s 10ms/step - loss: 0.8814
Epoch 92/200
1/1 [=====] - 0s 9ms/step - loss: 0.8753
Epoch 93/200
1/1 [=====] - 0s 10ms/step - loss: 0.8694
Epoch 94/200
1/1 [=====] - 0s 10ms/step - loss: 0.8641
Epoch 95/200
1/1 [=====] - 0s 9ms/step - loss: 0.8576
Epoch 96/200
1/1 [=====] - 0s 9ms/step - loss: 0.8508
Epoch 97/200
1/1 [=====] - 0s 10ms/step - loss: 0.8447
Epoch 98/200
1/1 [=====] - 0s 9ms/step - loss: 0.8384
Epoch 99/200
1/1 [=====] - 0s 9ms/step - loss: 0.8325
Epoch 100/200
```

```

1/1 [=====] - 0s 12ms/step - loss: 0.8270
Epoch 101/200
1/1 [=====] - 0s 12ms/step - loss: 0.8212
Epoch 102/200
1/1 [=====] - 0s 9ms/step - loss: 0.8166
Epoch 103/200
1/1 [=====] - 0s 9ms/step - loss: 0.8110
Epoch 104/200
1/1 [=====] - 0s 9ms/step - loss: 0.8060
Epoch 105/200
1/1 [=====] - 0s 9ms/step - loss: 0.8012
Epoch 106/200
1/1 [=====] - 0s 9ms/step - loss: 0.7963
Epoch 107/200
1/1 [=====] - 0s 9ms/step - loss: 0.7920
Epoch 108/200
1/1 [=====] - 0s 9ms/step - loss: 0.7873
Epoch 109/200
1/1 [=====] - 0s 9ms/step - loss: 0.7833
Epoch 110/200
1/1 [=====] - 0s 12ms/step - loss: 0.7787
Epoch 111/200
1/1 [=====] - 0s 9ms/step - loss: 0.7752
Epoch 112/200
1/1 [=====] - 0s 9ms/step - loss: 0.7708
Epoch 113/200
1/1 [=====] - 0s 9ms/step - loss: 0.7675
Epoch 114/200
1/1 [=====] - 0s 9ms/step - loss: 0.7634
Epoch 115/200
1/1 [=====] - 0s 9ms/step - loss: 0.7593
Epoch 116/200
1/1 [=====] - 0s 9ms/step - loss: 0.7564
Epoch 117/200
1/1 [=====] - 0s 8ms/step - loss: 0.7525
Epoch 118/200
1/1 [=====] - 0s 10ms/step - loss: 0.7498
Epoch 119/200
1/1 [=====] - 0s 7ms/step - loss: 0.7462
Epoch 120/200
1/1 [=====] - 0s 10ms/step - loss: 0.7438
Epoch 121/200
1/1 [=====] - 0s 10ms/step - loss: 0.7405
Epoch 122/200
1/1 [=====] - 0s 11ms/step - loss: 0.7390
Epoch 123/200
1/1 [=====] - 0s 9ms/step - loss: 0.7360
Epoch 124/200
1/1 [=====] - 0s 14ms/step - loss: 0.7350
Epoch 125/200
1/1 [=====] - 0s 9ms/step - loss: 0.7324
Epoch 126/200
1/1 [=====] - 0s 9ms/step - loss: 0.7306

```

```
Epoch 127/200
1/1 [=====] - 0s 8ms/step - loss: 0.7295
Epoch 128/200
1/1 [=====] - 0s 10ms/step - loss: 0.7293
Epoch 129/200
1/1 [=====] - 0s 9ms/step - loss: 0.7288
Epoch 130/200
1/1 [=====] - 0s 10ms/step - loss: 0.7307
Epoch 131/200
1/1 [=====] - 0s 9ms/step - loss: 0.7308
Epoch 132/200
1/1 [=====] - 0s 10ms/step - loss: 0.7353
Epoch 133/200
1/1 [=====] - 0s 9ms/step - loss: 0.7366
Epoch 134/200
1/1 [=====] - 0s 13ms/step - loss: 0.7442
Epoch 135/200
1/1 [=====] - 0s 12ms/step - loss: 0.7473
Epoch 136/200
1/1 [=====] - 0s 10ms/step - loss: 0.7610
Epoch 137/200
1/1 [=====] - 0s 8ms/step - loss: 0.7666
Epoch 138/200
1/1 [=====] - 0s 10ms/step - loss: 0.7889
Epoch 139/200
1/1 [=====] - 0s 8ms/step - loss: 0.7984
Epoch 140/200
1/1 [=====] - 0s 8ms/step - loss: 0.8313
Epoch 141/200
1/1 [=====] - 0s 8ms/step - loss: 0.8476
Epoch 142/200
1/1 [=====] - 0s 13ms/step - loss: 0.9021
Epoch 143/200
1/1 [=====] - 0s 13ms/step - loss: 0.9242
Epoch 144/200
1/1 [=====] - 0s 9ms/step - loss: 1.0144
Epoch 145/200
1/1 [=====] - 0s 8ms/step - loss: 1.0418
Epoch 146/200
1/1 [=====] - 0s 9ms/step - loss: 1.1601
Epoch 147/200
1/1 [=====] - 0s 13ms/step - loss: 1.1802
Epoch 148/200
1/1 [=====] - 0s 9ms/step - loss: 1.3158
Epoch 149/200
1/1 [=====] - 0s 8ms/step - loss: 1.3137
Epoch 150/200
1/1 [=====] - 0s 10ms/step - loss: 1.4663
Epoch 151/200
1/1 [=====] - 0s 9ms/step - loss: 1.4227
Epoch 152/200
1/1 [=====] - 0s 9ms/step - loss: 1.5643
Epoch 153/200
```

```

1/1 [=====] - 0s 8ms/step - loss: 1.4728
Epoch 154/200
1/1 [=====] - 0s 11ms/step - loss: 1.5047
Epoch 155/200
1/1 [=====] - 0s 8ms/step - loss: 1.2766
Epoch 156/200
1/1 [=====] - 0s 12ms/step - loss: 1.2891
Epoch 157/200
1/1 [=====] - 0s 9ms/step - loss: 1.1361
Epoch 158/200
1/1 [=====] - 0s 13ms/step - loss: 1.1534
Epoch 159/200
1/1 [=====] - 0s 8ms/step - loss: 1.0408
Epoch 160/200
1/1 [=====] - 0s 12ms/step - loss: 1.0738
Epoch 161/200
1/1 [=====] - 0s 8ms/step - loss: 1.0611
Epoch 162/200
1/1 [=====] - 0s 11ms/step - loss: 1.0828
Epoch 163/200
1/1 [=====] - 0s 7ms/step - loss: 1.0668
Epoch 164/200
1/1 [=====] - 0s 11ms/step - loss: 1.0904
Epoch 165/200
1/1 [=====] - 0s 10ms/step - loss: 0.9951
Epoch 166/200
1/1 [=====] - 0s 17ms/step - loss: 1.0311
Epoch 167/200
1/1 [=====] - 0s 10ms/step - loss: 1.0274
Epoch 168/200
1/1 [=====] - 0s 14ms/step - loss: 1.0544
Epoch 169/200
1/1 [=====] - 0s 7ms/step - loss: 1.0357
Epoch 170/200
1/1 [=====] - 0s 12ms/step - loss: 1.0626
Epoch 171/200
1/1 [=====] - 0s 13ms/step - loss: 1.0365
Epoch 172/200
1/1 [=====] - 0s 13ms/step - loss: 1.0616
Epoch 173/200
1/1 [=====] - 0s 8ms/step - loss: 0.9623
Epoch 174/200
1/1 [=====] - 0s 11ms/step - loss: 0.9939
Epoch 175/200
1/1 [=====] - 0s 11ms/step - loss: 0.9809
Epoch 176/200
1/1 [=====] - 0s 8ms/step - loss: 1.0022
Epoch 177/200
1/1 [=====] - 0s 13ms/step - loss: 0.9499
Epoch 178/200
1/1 [=====] - 0s 8ms/step - loss: 0.9658
Epoch 179/200
1/1 [=====] - 0s 12ms/step - loss: 0.8600

```

```

Epoch 180/200
1/1 [=====] - 0s 9ms/step - loss: 0.8813
Epoch 181/200
1/1 [=====] - 0s 12ms/step - loss: 0.8591
Epoch 182/200
1/1 [=====] - 0s 19ms/step - loss: 0.8662
Epoch 183/200
1/1 [=====] - 0s 11ms/step - loss: 0.8470
Epoch 184/200
1/1 [=====] - 0s 11ms/step - loss: 0.8412
Epoch 185/200
1/1 [=====] - 0s 10ms/step - loss: 0.8259
Epoch 186/200
1/1 [=====] - 0s 12ms/step - loss: 0.8331
Epoch 187/200
1/1 [=====] - 0s 13ms/step - loss: 0.7738
Epoch 188/200
1/1 [=====] - 0s 7ms/step - loss: 0.7953
Epoch 189/200
1/1 [=====] - 0s 7ms/step - loss: 0.7913
Epoch 190/200
1/1 [=====] - 0s 9ms/step - loss: 0.7911
Epoch 191/200
1/1 [=====] - 0s 7ms/step - loss: 0.7879
Epoch 192/200
1/1 [=====] - 0s 20ms/step - loss: 0.8067
Epoch 193/200
1/1 [=====] - 0s 8ms/step - loss: 0.8025
Epoch 194/200
1/1 [=====] - 0s 13ms/step - loss: 0.8278
Epoch 195/200
1/1 [=====] - 0s 14ms/step - loss: 0.8216
Epoch 196/200
1/1 [=====] - 0s 10ms/step - loss: 0.8300
Epoch 197/200
1/1 [=====] - 0s 8ms/step - loss: 0.7746
Epoch 198/200
1/1 [=====] - 0s 9ms/step - loss: 0.7891
Epoch 199/200
1/1 [=====] - 0s 7ms/step - loss: 0.7934
Epoch 200/200
1/1 [=====] - 0s 17ms/step - loss: 0.8365

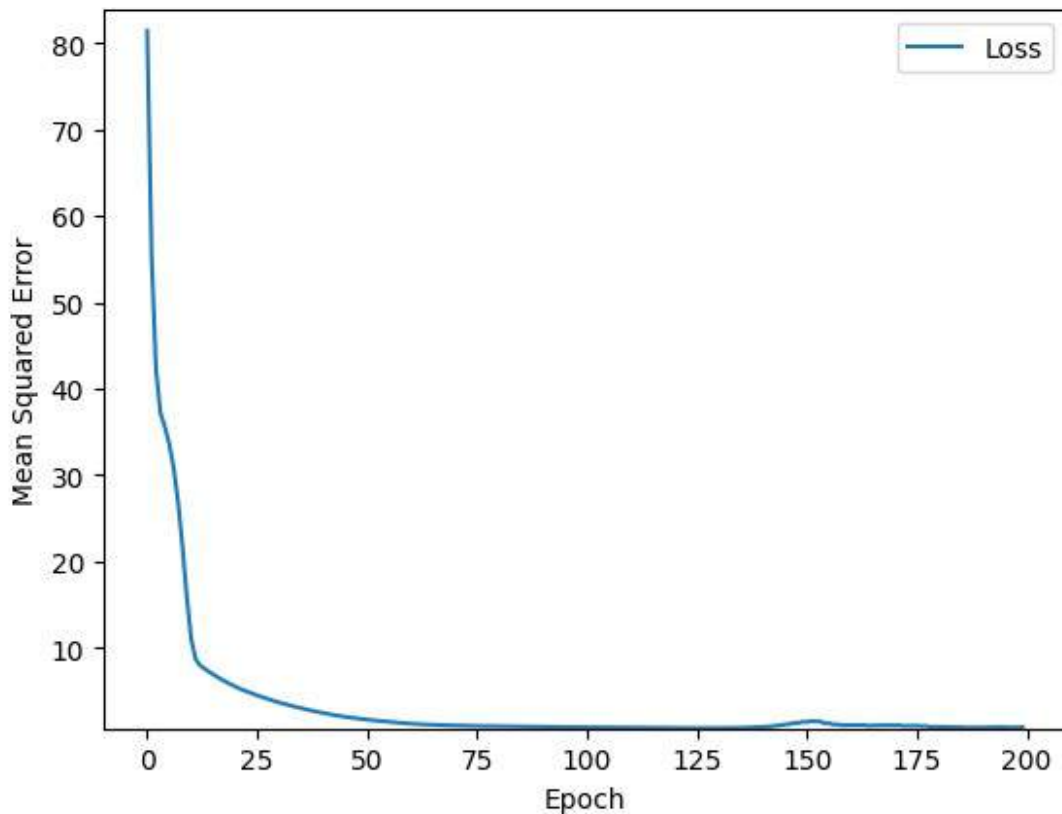
```

```

epochs = training_history.epoch
hist = pd.DataFrame(training_history.history)
rmse = hist["loss"]

plot_the_loss_curve(epochs, rmse)

```



So as you can see adding more neurons and layers in our case of small training dataset gives us the best result so far. Finally, let's try increasing the size of data points to 10000:

```
model = tf.keras.Sequential([
    keras.layers.Dense(units=8, activation='relu',
input_shape=[1]),
    keras.layers.Dense(units=8, activation='relu'),
    keras.layers.Dense(units=1)
])
model.compile(optimizer='adam', loss='mean_squared_error')
x = np.random.choice(np.arange(-10,10, .01), 10000)
fx = x ** 2

training_history = model.fit(x, fx, epochs=200)
```

Epoch 1/200
313/313 [=====] - 1s 2ms/step - loss: 1862.7976
Epoch 2/200
313/313 [=====] - 1s 2ms/step - loss: 565.2604
Epoch 3/200
313/313 [=====] - 1s 2ms/step - loss: 133.3320
Epoch 4/200
313/313 [=====] - 1s 2ms/step - loss:


```
98.7223
Epoch 5/200
313/313 [=====] - 1s 2ms/step - loss:
75.6769
Epoch 6/200
313/313 [=====] - 1s 2ms/step - loss:
60.3313
Epoch 7/200
313/313 [=====] - 1s 2ms/step - loss:
51.0604
Epoch 8/200
313/313 [=====] - 1s 2ms/step - loss:
45.6382
Epoch 9/200
313/313 [=====] - 1s 2ms/step - loss:
41.7288
Epoch 10/200
313/313 [=====] - 1s 2ms/step - loss:
38.0364
Epoch 11/200
313/313 [=====] - 1s 2ms/step - loss:
34.3731
Epoch 12/200
313/313 [=====] - 1s 2ms/step - loss:
30.6229
Epoch 13/200
313/313 [=====] - 1s 2ms/step - loss:
27.0908
Epoch 14/200
313/313 [=====] - 1s 2ms/step - loss:
23.4833
Epoch 15/200
313/313 [=====] - 1s 3ms/step - loss:
19.9093
Epoch 16/200
313/313 [=====] - 1s 3ms/step - loss:
16.4108
Epoch 17/200
313/313 [=====] - 1s 2ms/step - loss:
13.1163
Epoch 18/200
313/313 [=====] - 1s 2ms/step - loss:
10.2349
Epoch 19/200
313/313 [=====] - 1s 2ms/step - loss:
7.8862
Epoch 20/200
313/313 [=====] - 1s 2ms/step - loss:
5.9571
Epoch 21/200
313/313 [=====] - 1s 2ms/step - loss:
4.5392
Epoch 22/200
```

```
313/313 [=====] - 1s 2ms/step - loss:
3.5195
Epoch 23/200
313/313 [=====] - 1s 2ms/step - loss:
2.8526
Epoch 24/200
313/313 [=====] - 1s 2ms/step - loss:
2.4488
Epoch 25/200
313/313 [=====] - 1s 2ms/step - loss:
2.2002
Epoch 26/200
313/313 [=====] - 1s 2ms/step - loss:
2.0182
Epoch 27/200
313/313 [=====] - 1s 2ms/step - loss:
1.8768
Epoch 28/200
313/313 [=====] - 1s 2ms/step - loss:
1.7464
Epoch 29/200
313/313 [=====] - 1s 2ms/step - loss:
1.6512
Epoch 30/200
313/313 [=====] - 1s 2ms/step - loss:
1.5371
Epoch 31/200
313/313 [=====] - 1s 2ms/step - loss:
1.4682
Epoch 32/200
313/313 [=====] - 1s 2ms/step - loss:
1.3951
Epoch 33/200
313/313 [=====] - 1s 2ms/step - loss:
1.3191
Epoch 34/200
313/313 [=====] - 1s 2ms/step - loss:
1.2666
Epoch 35/200
313/313 [=====] - 1s 2ms/step - loss:
1.2124
Epoch 36/200
313/313 [=====] - 1s 3ms/step - loss:
1.1471
Epoch 37/200
313/313 [=====] - 1s 2ms/step - loss:
1.1165
Epoch 38/200
313/313 [=====] - 1s 2ms/step - loss:
1.0610
Epoch 39/200
313/313 [=====] - 1s 2ms/step - loss:
1.0162
```

```
Epoch 40/200
313/313 [=====] - 1s 2ms/step - loss:
0.9738
Epoch 41/200
313/313 [=====] - 1s 2ms/step - loss:
0.9558
Epoch 42/200
313/313 [=====] - 1s 2ms/step - loss:
0.9141
Epoch 43/200
313/313 [=====] - 1s 2ms/step - loss:
0.8729
Epoch 44/200
313/313 [=====] - 1s 2ms/step - loss:
0.8355
Epoch 45/200
313/313 [=====] - 1s 2ms/step - loss:
0.8088
Epoch 46/200
313/313 [=====] - 1s 2ms/step - loss:
0.7652
Epoch 47/200
313/313 [=====] - 1s 2ms/step - loss:
0.7008
Epoch 48/200
313/313 [=====] - 1s 2ms/step - loss:
0.6764
Epoch 49/200
313/313 [=====] - 1s 2ms/step - loss:
0.6303
Epoch 50/200
313/313 [=====] - 1s 2ms/step - loss:
0.6113
Epoch 51/200
313/313 [=====] - 1s 2ms/step - loss:
0.5908
Epoch 52/200
313/313 [=====] - 1s 4ms/step - loss:
0.5715
Epoch 53/200
313/313 [=====] - 1s 3ms/step - loss:
0.5429
Epoch 54/200
313/313 [=====] - 1s 2ms/step - loss:
0.5184
Epoch 55/200
313/313 [=====] - 1s 2ms/step - loss:
0.5138
Epoch 56/200
313/313 [=====] - 1s 3ms/step - loss:
0.4951
Epoch 57/200
313/313 [=====] - 1s 3ms/step - loss:
```

```
0.4746
Epoch 58/200
313/313 [=====] - 1s 3ms/step - loss:
0.4631
Epoch 59/200
313/313 [=====] - 1s 4ms/step - loss:
0.4558
Epoch 60/200
313/313 [=====] - 2s 5ms/step - loss:
0.4528
Epoch 61/200
313/313 [=====] - 1s 2ms/step - loss:
0.4448
Epoch 62/200
313/313 [=====] - 1s 2ms/step - loss:
0.4324
Epoch 63/200
313/313 [=====] - 1s 2ms/step - loss:
0.4265
Epoch 64/200
313/313 [=====] - 1s 2ms/step - loss:
0.4146
Epoch 65/200
313/313 [=====] - 1s 2ms/step - loss:
0.4114
Epoch 66/200
313/313 [=====] - 1s 2ms/step - loss:
0.4085
Epoch 67/200
313/313 [=====] - 1s 2ms/step - loss:
0.4101
Epoch 68/200
313/313 [=====] - 1s 2ms/step - loss:
0.4051
Epoch 69/200
313/313 [=====] - 1s 2ms/step - loss:
0.3863
Epoch 70/200
313/313 [=====] - 1s 2ms/step - loss:
0.3979
Epoch 71/200
313/313 [=====] - 1s 2ms/step - loss:
0.3838
Epoch 72/200
313/313 [=====] - 1s 2ms/step - loss:
0.3846
Epoch 73/200
313/313 [=====] - 1s 2ms/step - loss:
0.3804
Epoch 74/200
313/313 [=====] - 1s 3ms/step - loss:
0.3749
Epoch 75/200
```

```
313/313 [=====] - 1s 3ms/step - loss:
0.3779
Epoch 76/200
313/313 [=====] - 1s 3ms/step - loss:
0.3775
Epoch 77/200
313/313 [=====] - 1s 2ms/step - loss:
0.3750
Epoch 78/200
313/313 [=====] - 1s 2ms/step - loss:
0.3667
Epoch 79/200
313/313 [=====] - 1s 2ms/step - loss:
0.3642
Epoch 80/200
313/313 [=====] - 1s 2ms/step - loss:
0.3695
Epoch 81/200
313/313 [=====] - 1s 2ms/step - loss:
0.3655
Epoch 82/200
313/313 [=====] - 1s 2ms/step - loss:
0.3516
Epoch 83/200
313/313 [=====] - 1s 2ms/step - loss:
0.3549
Epoch 84/200
313/313 [=====] - 1s 2ms/step - loss:
0.3511
Epoch 85/200
313/313 [=====] - 1s 2ms/step - loss:
0.3519
Epoch 86/200
313/313 [=====] - 1s 2ms/step - loss:
0.3504
Epoch 87/200
313/313 [=====] - 1s 2ms/step - loss:
0.3562
Epoch 88/200
313/313 [=====] - 1s 2ms/step - loss:
0.3487
Epoch 89/200
313/313 [=====] - 1s 2ms/step - loss:
0.3454
Epoch 90/200
313/313 [=====] - 1s 2ms/step - loss:
0.3358
Epoch 91/200
313/313 [=====] - 1s 2ms/step - loss:
0.3433
Epoch 92/200
313/313 [=====] - 1s 2ms/step - loss:
0.3356
```

```
Epoch 93/200
313/313 [=====] - 1s 2ms/step - loss:
0.3367
Epoch 94/200
313/313 [=====] - 1s 2ms/step - loss:
0.3329
Epoch 95/200
313/313 [=====] - 1s 2ms/step - loss:
0.3375
Epoch 96/200
313/313 [=====] - 1s 2ms/step - loss:
0.3345
Epoch 97/200
313/313 [=====] - 1s 2ms/step - loss:
0.3332
Epoch 98/200
313/313 [=====] - 1s 2ms/step - loss:
0.3255
Epoch 99/200
313/313 [=====] - 1s 2ms/step - loss:
0.3321
Epoch 100/200
313/313 [=====] - 1s 2ms/step - loss:
0.3223
Epoch 101/200
313/313 [=====] - 1s 2ms/step - loss:
0.3306
Epoch 102/200
313/313 [=====] - 1s 2ms/step - loss:
0.3240
Epoch 103/200
313/313 [=====] - 1s 2ms/step - loss:
0.3132
Epoch 104/200
313/313 [=====] - 1s 2ms/step - loss:
0.3249
Epoch 105/200
313/313 [=====] - 1s 2ms/step - loss:
0.3216
Epoch 106/200
313/313 [=====] - 1s 2ms/step - loss:
0.3159
Epoch 107/200
313/313 [=====] - 1s 2ms/step - loss:
0.3205
Epoch 108/200
313/313 [=====] - 1s 2ms/step - loss:
0.3236
Epoch 109/200
313/313 [=====] - 1s 2ms/step - loss:
0.3148
Epoch 110/200
313/313 [=====] - 1s 2ms/step - loss:
```

```
0.3126
Epoch 111/200
313/313 [=====] - 1s 2ms/step - loss:
0.3069
Epoch 112/200
313/313 [=====] - 1s 2ms/step - loss:
0.3167
Epoch 113/200
313/313 [=====] - 1s 2ms/step - loss:
0.3134
Epoch 114/200
313/313 [=====] - 1s 2ms/step - loss:
0.3085
Epoch 115/200
313/313 [=====] - 1s 2ms/step - loss:
0.3110
Epoch 116/200
313/313 [=====] - 1s 2ms/step - loss:
0.3029
Epoch 117/200
313/313 [=====] - 1s 3ms/step - loss:
0.3039
Epoch 118/200
313/313 [=====] - 1s 3ms/step - loss:
0.3027
Epoch 119/200
313/313 [=====] - 1s 2ms/step - loss:
0.3060
Epoch 120/200
313/313 [=====] - 1s 2ms/step - loss:
0.3090
Epoch 121/200
313/313 [=====] - 1s 2ms/step - loss:
0.2989
Epoch 122/200
313/313 [=====] - 1s 2ms/step - loss:
0.3017
Epoch 123/200
313/313 [=====] - 1s 2ms/step - loss:
0.2952
Epoch 124/200
313/313 [=====] - 1s 2ms/step - loss:
0.2871
Epoch 125/200
313/313 [=====] - 1s 2ms/step - loss:
0.2941
Epoch 126/200
313/313 [=====] - 1s 2ms/step - loss:
0.2921
Epoch 127/200
313/313 [=====] - 1s 2ms/step - loss:
0.2874
Epoch 128/200
```

```

313/313 [=====] - 1s 2ms/step - loss:
0.2956
Epoch 129/200
313/313 [=====] - 1s 2ms/step - loss:
0.2874
Epoch 130/200
313/313 [=====] - 1s 2ms/step - loss:
0.2872
Epoch 131/200
313/313 [=====] - 1s 2ms/step - loss:
0.2916
Epoch 132/200
313/313 [=====] - 1s 2ms/step - loss:
0.2825
Epoch 133/200
313/313 [=====] - 1s 2ms/step - loss:
0.2832
Epoch 134/200
313/313 [=====] - 1s 2ms/step - loss:
0.2883
Epoch 135/200
313/313 [=====] - 1s 2ms/step - loss:
0.2806
Epoch 136/200
313/313 [=====] - 1s 2ms/step - loss:
0.2785
Epoch 137/200
313/313 [=====] - 1s 2ms/step - loss:
0.2766
Epoch 138/200
313/313 [=====] - 1s 2ms/step - loss:
0.2810
Epoch 139/200
313/313 [=====] - 1s 2ms/step - loss:
0.2776
Epoch 140/200
313/313 [=====] - 1s 3ms/step - loss:
0.2717
Epoch 141/200
313/313 [=====] - 1s 2ms/step - loss:
0.2700
Epoch 142/200
313/313 [=====] - 1s 2ms/step - loss:
0.2748
Epoch 143/200
313/313 [=====] - 1s 2ms/step - loss:
0.2789
Epoch 144/200
313/313 [=====] - 1s 2ms/step - loss:
0.2690
Epoch 145/200
313/313 [=====] - 1s 2ms/step - loss:
0.2722

```



```
Epoch 146/200
313/313 [=====] - 1s 2ms/step - loss:
0.2766
Epoch 147/200
313/313 [=====] - 1s 2ms/step - loss:
0.2729
Epoch 148/200
313/313 [=====] - 1s 2ms/step - loss:
0.2676
Epoch 149/200
313/313 [=====] - 1s 2ms/step - loss:
0.2659
Epoch 150/200
313/313 [=====] - 1s 2ms/step - loss:
0.2635
Epoch 151/200
313/313 [=====] - 1s 2ms/step - loss:
0.2609
Epoch 152/200
313/313 [=====] - 1s 2ms/step - loss:
0.2589
Epoch 153/200
313/313 [=====] - 1s 2ms/step - loss:
0.2620
Epoch 154/200
313/313 [=====] - 1s 2ms/step - loss:
0.2589
Epoch 155/200
313/313 [=====] - 1s 2ms/step - loss:
0.2636
Epoch 156/200
313/313 [=====] - 1s 2ms/step - loss:
0.2661
Epoch 157/200
313/313 [=====] - 1s 2ms/step - loss:
0.2544
Epoch 158/200
313/313 [=====] - 1s 2ms/step - loss:
0.2507
Epoch 159/200
313/313 [=====] - 1s 2ms/step - loss:
0.2530
Epoch 160/200
313/313 [=====] - 1s 3ms/step - loss:
0.2554
Epoch 161/200
313/313 [=====] - 1s 2ms/step - loss:
0.2537
Epoch 162/200
313/313 [=====] - 1s 3ms/step - loss:
0.2569
Epoch 163/200
313/313 [=====] - 1s 2ms/step - loss:
```

```
0.2491
Epoch 164/200
313/313 [=====] - 1s 2ms/step - loss:
0.2530
Epoch 165/200
313/313 [=====] - 1s 2ms/step - loss:
0.2466
Epoch 166/200
313/313 [=====] - 1s 2ms/step - loss:
0.2414
Epoch 167/200
313/313 [=====] - 1s 2ms/step - loss:
0.2541
Epoch 168/200
313/313 [=====] - 1s 2ms/step - loss:
0.2465
Epoch 169/200
313/313 [=====] - 1s 2ms/step - loss:
0.2423
Epoch 170/200
313/313 [=====] - 1s 2ms/step - loss:
0.2456
Epoch 171/200
313/313 [=====] - 1s 2ms/step - loss:
0.2425
Epoch 172/200
313/313 [=====] - 1s 2ms/step - loss:
0.2450
Epoch 173/200
313/313 [=====] - 1s 2ms/step - loss:
0.2453
Epoch 174/200
313/313 [=====] - 1s 2ms/step - loss:
0.2416
Epoch 175/200
313/313 [=====] - 1s 2ms/step - loss:
0.2391
Epoch 176/200
313/313 [=====] - 1s 2ms/step - loss:
0.2360
Epoch 177/200
313/313 [=====] - 1s 2ms/step - loss:
0.2387
Epoch 178/200
313/313 [=====] - 1s 2ms/step - loss:
0.2344
Epoch 179/200
313/313 [=====] - 1s 2ms/step - loss:
0.2321
Epoch 180/200
313/313 [=====] - 1s 2ms/step - loss:
0.2382
Epoch 181/200
313/313 [=====] - 1s 2ms/step - loss:
```

```
0.2351
Epoch 182/200
313/313 [=====] - 1s 2ms/step - loss:
0.2359
Epoch 183/200
313/313 [=====] - 1s 2ms/step - loss:
0.2265
Epoch 184/200
313/313 [=====] - 1s 3ms/step - loss:
0.2352
Epoch 185/200
313/313 [=====] - 1s 2ms/step - loss:
0.2300
Epoch 186/200
313/313 [=====] - 1s 2ms/step - loss:
0.2299
Epoch 187/200
313/313 [=====] - 1s 2ms/step - loss:
0.2276
Epoch 188/200
313/313 [=====] - 1s 2ms/step - loss:
0.2332
Epoch 189/200
313/313 [=====] - 1s 2ms/step - loss:
0.2225
Epoch 190/200
313/313 [=====] - 1s 2ms/step - loss:
0.2291
Epoch 191/200
313/313 [=====] - 1s 2ms/step - loss:
0.2218
Epoch 192/200
313/313 [=====] - 1s 2ms/step - loss:
0.2301
Epoch 193/200
313/313 [=====] - 1s 2ms/step - loss:
0.2224
Epoch 194/200
313/313 [=====] - 1s 2ms/step - loss:
0.2221
Epoch 195/200
313/313 [=====] - 1s 2ms/step - loss:
0.2219
Epoch 196/200
313/313 [=====] - 1s 2ms/step - loss:
0.2231
Epoch 197/200
313/313 [=====] - 1s 2ms/step - loss:
0.2213
Epoch 198/200
313/313 [=====] - 1s 2ms/step - loss:
0.2158
Epoch 199/200
```

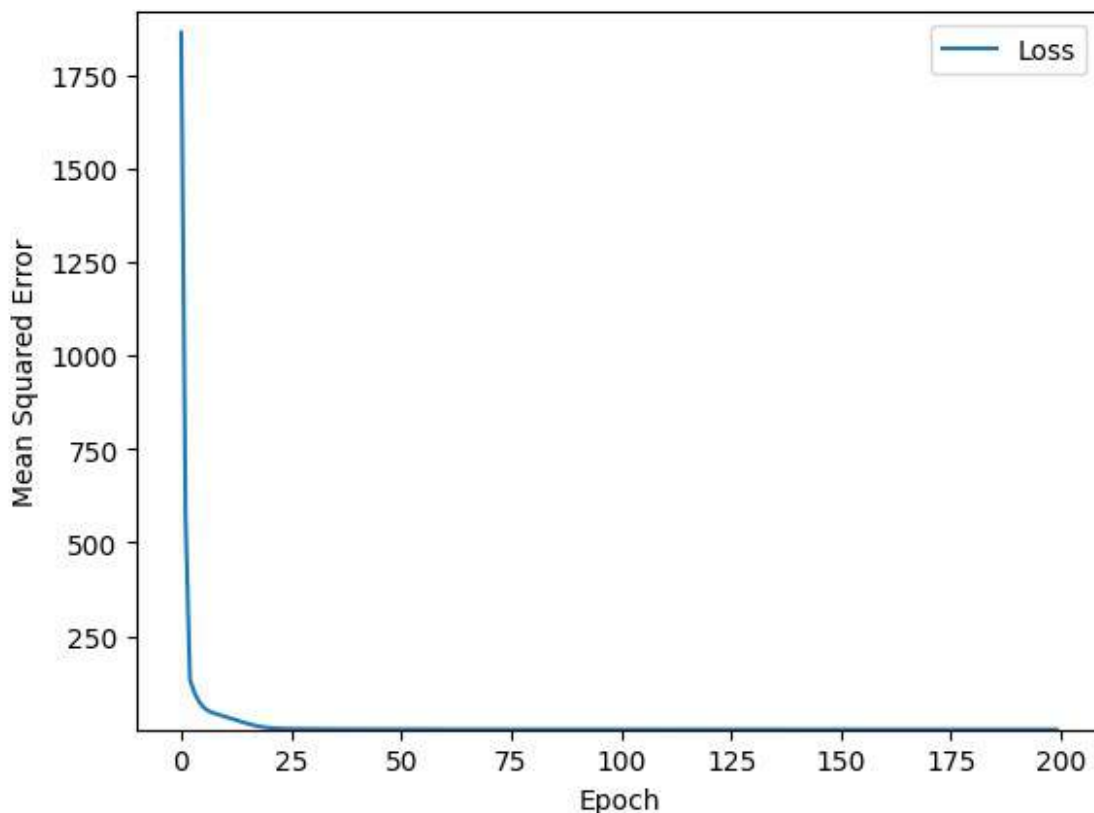
```

313/313 [=====] - 1s 2ms/step - loss:
0.2258
Epoch 200/200
313/313 [=====] - 1s 2ms/step - loss:
0.2197

epochs = training_history.epoch
hist = pd.DataFrame(training_history.history)
rmse = hist["loss"]

plot_the_loss_curve(epochs, rmse)

```



So as you can see, giving it more examples is one of the best solutions you can use to allow the model to better *learn*.

You might have thought it's 31? But it ended up being close but not exact. Why do you think that is?

Remember that neural networks deal with probabilities, so given the data that we fed the network with, it calculated that there is a very high probability that the relationship between x and $f(x)$ is $f(x)=3*x+1$, but with only 6 data points we can't know for sure. As a result, the result for 10 is very close to 31, but not necessarily 31.

As you work with neural networks, you'll see this pattern recurring. You will almost always deal with probabilities, not certainties, and will do a little bit of coding to figure out what the result is based on the probabilities, particularly when it comes to classification.

2. Multi-layer Perceptrons

2.1. Introduction to MLP

Multi-layer Perceptrons, or **MLPs**, are a type of feedforward neural network with one or more hidden layers between the input and output layers. MLPs are designed to tackle complex problems by learning intricate patterns in data. Each neuron in an MLP is a perceptron that performs weighted sum and activation functions on the inputs.

An MLP architecture consists of an input layer, one or more hidden layers, and an output layer. Each neuron in a hidden layer connects to every neuron in the previous and subsequent layers. This dense connectivity allows MLPs to learn intricate relationships in data.

Forward Propagation: During forward propagation, data flows through the network from the input layer to the output layer. Each neuron computes the weighted sum of its inputs, applies an activation function, and passes the result to the next layer.

Backpropagation: Backpropagation is the process of updating the weights in the network to minimize the loss function. It involves propagating the error backward from the output layer to the input layer and adjusting the weights using gradient descent. The gradients of the loss with respect to each weight are computed, and the weights are updated accordingly.

Hyperparameter tuning is essential to achieve the best performance from an MLP. Key hyperparameters include the learning rate, the number of hidden layers, the number of neurons in each layer, activation functions, and regularization parameters. Tuning these hyperparameters can significantly impact the model's convergence and generalization capabilities.

Python Code Examples:

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import to_categorical
from sklearn.model_selection import train_test_split

# Generate some sample data (replace with your actual data)
num_samples = 1000
num_features = 10
num_classes = 3

X = np.random.rand(num_samples, num_features)
y = np.random.randint(num_classes, size=num_samples)

# Convert labels to one-hot encoded format
y_onehot = to_categorical(y, num_classes=num_classes)

# Split data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y_onehot,
test_size=0.2, random_state=42)

# Build the MLP model
model = Sequential()
model.add(Dense(units=64, activation='relu',
```

```

input_dim=num_features))
model.add(Dense(units=128, activation='relu'))
model.add(Dense(units=64, activation='relu'))
model.add(Dense(units=num_classes, activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])

# Train the model
num_epochs = 10
batch_size = 32
model.fit(X_train, y_train, epochs=num_epochs,
batch_size=batch_size, validation_data=(X_val, y_val))

```

In this example:

- We generate sample data (X and y) for demonstration purposes. Replace this with your actual dataset.
- We convert the integer labels (y) into one-hot encoded format using `to_categorical` from **Keras**.
- The data is split into training and validation sets using `train_test_split` from **scikit-learn**.
- The MLP model is built using the **Sequential** API in **Keras**, with specified hidden layers and activation functions.
- The model is compiled with categorical *cross-entropy loss* and the *Adam optimizer*.
- The model is trained using the `fit` method, with the training data and validation data provided.

Remember to replace the sample data and labels with your actual dataset, and adjust the number of features, classes, and other parameters according to your problem. This example should give you a basic structure for creating and training an MLP using Keras.

```

from IPython.display import Image
%matplotlib inline

from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive
%cd /content/drive/My Drive/Colab Notebooks/AI Course/
/content/drive/My Drive/Colab Notebooks/SoftServe AI

```

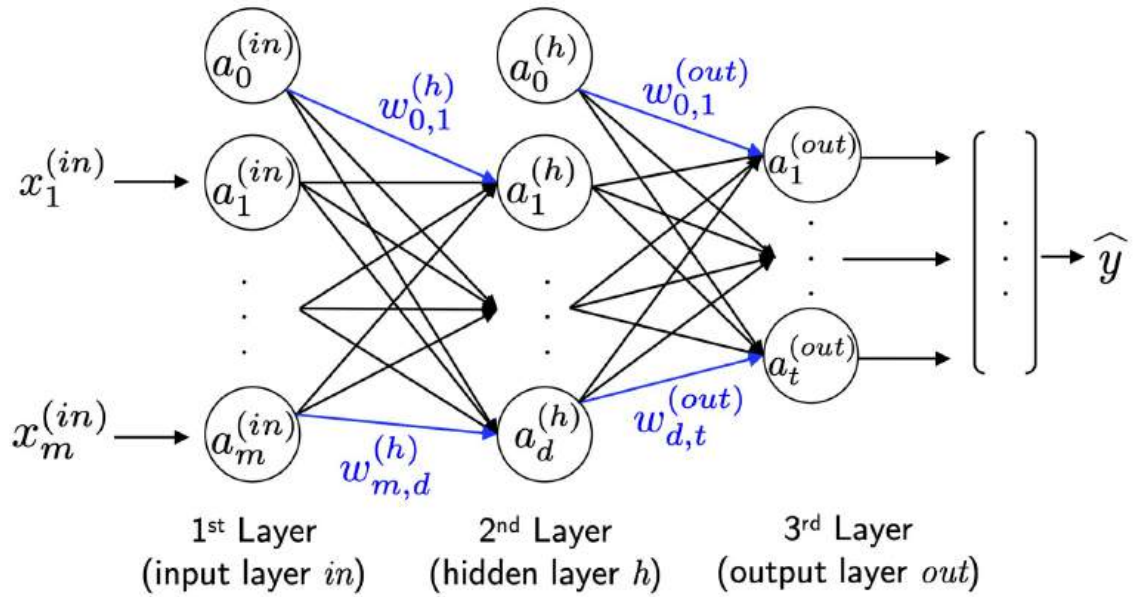
###2.2. Multilayer Neural Network architecture

The following figure illustrates the concept of an MLP consisting of three layers:

```

Image(filename='images/12_02.png', width=600)

```



The MLP depicted in the preceding figure has one input layer, one hidden layer, and one output layer. The units in the hidden layer are fully connected to the input layer, and the output layer is fully connected to the hidden layer. If such a network has more than one hidden layer, we also call it a **deep artificial NN**.

As shown in the preceding figure, we denote the i th activation unit in the l th layer as $a_i^{(l)}$. To make the math and code implementations a bit more intuitive, we will not use numerical indices to refer to layers, but we will use the *in* superscript for the input layer, the *h* superscript for the hidden layer, and the *out* superscript for the output layer. For instance, $a_i^{(in)}$ refers to the i th value in the input layer, $a_i^{(h)}$ refers to the i th unit in the hidden layer, and $a_i^{(out)}$ refers to the i th unit in the output layer. Here, the activation units $a_0^{(in)}$ and $a_0^{(h)}$ are the *bias* units, which we set equal to 1.

The activation of the units in the input layer is just its input plus the bias unit:

$$a^{(in)} = \begin{bmatrix} a_0^{(in)} \\ a_1^{(in)} \\ \vdots \\ a_m^{(in)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(in)} \\ \vdots \\ x_m^{(in)} \end{bmatrix}$$

Each unit in layer l is connected to all units in layer $l+1$ via a weight coefficient. For example, the connection between the k th unit in layer l to the j th unit in layer $l+1$ will be written as $w_{k,j}^{(l)}$. Referring back to the previous figure, we denote the weight matrix that connects the input to the hidden layer as $w^{(h)}$, and we write the matrix that connects the hidden layer to the output layer as $w^{(out)}$.

While one unit in the output layer would suffice for a binary classification task, we saw a more general form of NN in the preceding figure, which allows us to perform multiclass classification.

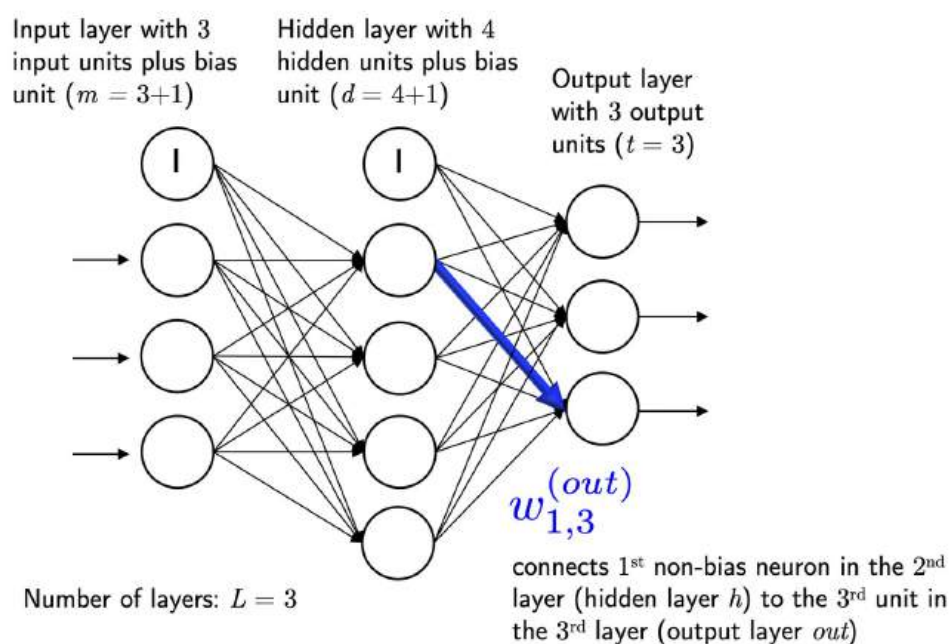
For example, we can encode the three class labels in the familiar **Iris dataset** ($0 = \textit{Setosa}$, $1 = \textit{Versicolor}$, $2 = \textit{Virginica}$) as follows:

$$0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad 1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad 2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

This one-hot vector representation allows us to tackle classification tasks with an arbitrary number of unique class labels present in the training dataset.

Let's summarize what we have just learned in a descriptive illustration of a simplified 3-4-3 MLP:

```
Image(filename='images/12_03.png', width=500)
```



2.3. Activating a neural network via forward propagation

To understand how **forward propagation** fits into the context of learning an MLP model, let's summarize the MLP learning procedure in three simple steps:

1. Starting at the input layer, we forward propagate the patterns of the training data through the network to generate an output.
2. Based on the network's output, we calculate the error that we want to minimize using a cost function that we will describe later.
3. We backpropagate the error, find its derivative with respect to each weight in the network, and update the model.

Finally, after we repeat these three steps for multiple epochs and learn the weights of the MLP, we use forward propagation to calculate the network output and apply a threshold function to obtain the predicted class labels in the one-hot representation, which we described in the previous section.

MLP is a typical example of a feedforward artificial NN. The term *feedforward* refers to the fact that each layer serves as the input to the next layer without loops, in contrast to recurrent NNs—an architecture. The term *multilayer perceptron* may sound a little bit confusing since the artificial neurons in this network architecture are typically sigmoid units, not perceptrons. We can think of the neurons in the MLP as logistic regression units that return values in the continuous range between 0 and 1.

Classifying handwritten digits

In this section, we will implement and train multilayer NN to classify handwritten digits from the popular **Mixed National Institute of Standards and Technology (MNIST) dataset** that has been constructed by Yann LeCun and others and serves as a popular benchmark dataset for machine learning algorithms (*Gradient-Based Learning Applied to Document Recognition*, Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, *Proceedings of the IEEE*, 86(11): 2278-2324, November 1998).

The MNIST dataset is publicly available at <http://yann.lecun.com/exdb/mnist/> and consists of the following four parts:

- Training dataset images: train-images-idx3-ubyte.gz (9.9 MB, 47 MB unzipped, and 60,000 examples)
- Training dataset labels: train-labels-idx1-ubyte.gz (29 KB, 60 KB unzipped, and 60,000 labels)
- Test dataset images: t10k-images-idx3-ubyte.gz (1.6 MB, 7.8 MB unzipped, and 10,000 examples)
- Test dataset labels: t10k-labels-idx1-ubyte.gz (5 KB, 10 KB unzipped, and 10,000 labels)

The MNIST dataset was constructed from two datasets of the US National Institute of Standards and Technology (NIST). The training dataset consists of handwritten digits from 250 different people, 50 percent high school students and 50 percent employees from the Census Bureau. Note that the test dataset contains handwritten digits from different people following the same split.

In this section, we will only be working with a subset of MNIST, thus, we only need to download the training set images and training set labels.

After downloading the files, simply run the next code cell to unzip the files.

```
# this code cell unzips mnist

import sys
import gzip
import shutil
import os

if (sys.version_info > (3, 0)):
    writemode = 'wb'
else:
    writemode = 'w'

zipped_mnist = [f for f in os.listdir() if f.endswith('ubyte.gz')]
for z in zipped_mnist:
    with gzip.GzipFile(z, mode='rb') as decompressed, open(z[:-3],
```

```
writemode) as outfile:
    outfile.write(decompressed.read())
```

IGNORE IF THE CODE CELL ABOVE EXECUTED WITHOUT PROBLEMS:

If you have issues with the code cell above, I recommend unzipping the files using the Unix/Linux gzip tool from the terminal for efficiency, e.g., using the command

```
gzip *ubyte.gz -d
```

in your local MNIST download directory, or, using your favorite unzipping tool if you are working with a machine running on Microsoft Windows. The images are stored in byte form, and using the following function, we will read them into NumPy arrays that we will use to train our MLP.

Please note that if you are **not** using gzip, please make sure the files are named

- train-images-idx3-ubyte
- train-labels-idx1-ubyte
- t10k-images-idx3-ubyte
- t10k-labels-idx1-ubyte

If a file is e.g., named `train-images.idx3-ubyte` after unzipping (this is due to the fact that certain tools try to guess a file suffix), please rename it to `train-images-idx3-ubyte` before proceeding.

The images are stored in byte format, and we will read them into NumPy arrays that we will use to train and test our MLP implementation. In order to do that, we will define the following helper function:

```
import os
import struct
import numpy as np

def load_mnist(path, kind='train'):
    """Load MNIST data from `path`"""
    labels_path = os.path.join(path,
                                '%s-labels.idx1-ubyte' % kind)
    images_path = os.path.join(path,
                                '%s-images.idx3-ubyte' % kind)

    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',
                                lbpath.read(8))
        labels = np.fromfile(lbpath,
                              dtype=np.uint8)

    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack(">IIII",
                                                imgpath.read(16))
```

```

        images = np.fromfile(imgpath,
                               dtype=np.uint8).reshape(len(labels),
784)
        images = ((images / 255.) - .5) * 2

    return images, labels

```

The `load_mnist` function returns two arrays, the first being an $n \times m$ dimensional NumPy array (images), where n is the number of examples and m is the number of features (here, pixels). The training dataset consists of 60,000 training digits and the test dataset contains 10,000 examples, respectively.

The images in the MNIST dataset consist of 28×28 pixels, and each pixel is represented by a grayscale intensity value. Here, we unroll the 28×28 pixels into one-dimensional row vectors, which represent the rows in our images array (784 per row or image). The second array (labels) returned by the `load_mnist` function contains the corresponding target variable, the class labels (integers 0-9) of the handwritten digits.

The way we read in the image might seem a little bit strange at first:

```

magic, n = struct.unpack('>II', lbpath.read(8))
labels = np.fromfile(lbpath, dtype=np.uint8)

```

To understand how those two lines of code work, let's take a look at the dataset description from the MNIST website:

```
Image(filename='images/12_14.png', width=500)
```

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000801(2049)	magic number (MSB first)
0004	32 bit integer	60000	number of items
0008	unsigned byte	??	label
0009	unsigned byte	??	label
.....			
xxxx	unsigned byte	??	label

Using the two preceding lines of code, we first read in the magic number, which is a description of the file protocol, as well as the number of items (n) from the file buffer, before we load the following bytes into a NumPy array using the `fromfile` method. The `fmt` parameter value, `>II`, that we passed as an argument to `struct.unpack` can be composed into the two following parts:

- `>` This is *big-endian*—it defines the order in which a sequence of bytes is stored (if you are unfamiliar with the terms big-endian and little-endian, you can find an excellent article about [Endianness](#) on Wikipedia);
- `I` This is an unsigned integer.

Finally, we also normalized the pixels values in MNIST to the range -1 to 1 (originally 0 to 255) via the following code line:

```
images = ((images / 255.) - .5) * 2
```

The reason behind this is that *gradient-based optimization* is much more stable under these conditions. Note that we scaled the images on a pixel-by-pixel basis, which is different from the feature scaling approach.

By executing the following code, we will now load the 60,000 training instances as well as the 10,000 test examples from the local directory where we unzipped the MNIST dataset. (In the following code snippet, it is assumed that the downloaded MNIST files were unzipped to the same directory in which this code was executed.)

```
!ls

X_train, y_train = load_mnist('', kind='train')
# /content/drive/MyDrive/Colab Notebooks/SoftServe AI/train-
labels.idx1-ubyte
print('Rows: %d, columns: %d' % (X_train.shape[0],
X_train.shape[1]))

Rows: 60000, columns: 784

X_test, y_test = load_mnist('', kind='t10k')
print('Rows: %d, columns: %d' % (X_test.shape[0], X_test.shape[1]))

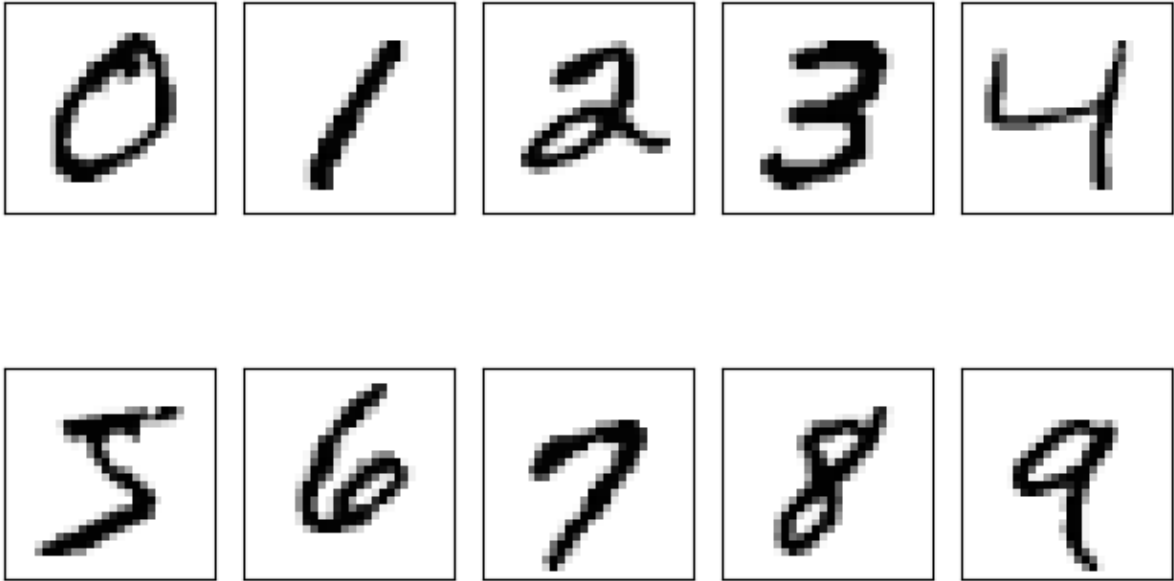
Rows: 10000, columns: 784
```

To get an idea of how those images in MNIST look, let's visualize examples of the digits 0 – 9 after reshaping the 784-pixel vectors from our feature matrix into the original 28×28 image that we can plot via **Matplotlib's** `imshow` function:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(nrows=2, ncols=5, sharex=True, sharey=True)
ax = ax.flatten()
for i in range(10):
    img = X_train[y_train == i][0].reshape(28, 28)
    ax[i].imshow(img, cmap='Greys')

ax[0].set_xticks([])
ax[0].set_yticks([])
plt.tight_layout()
# plt.savefig('images/12_5.png', dpi=300)
plt.show()
```



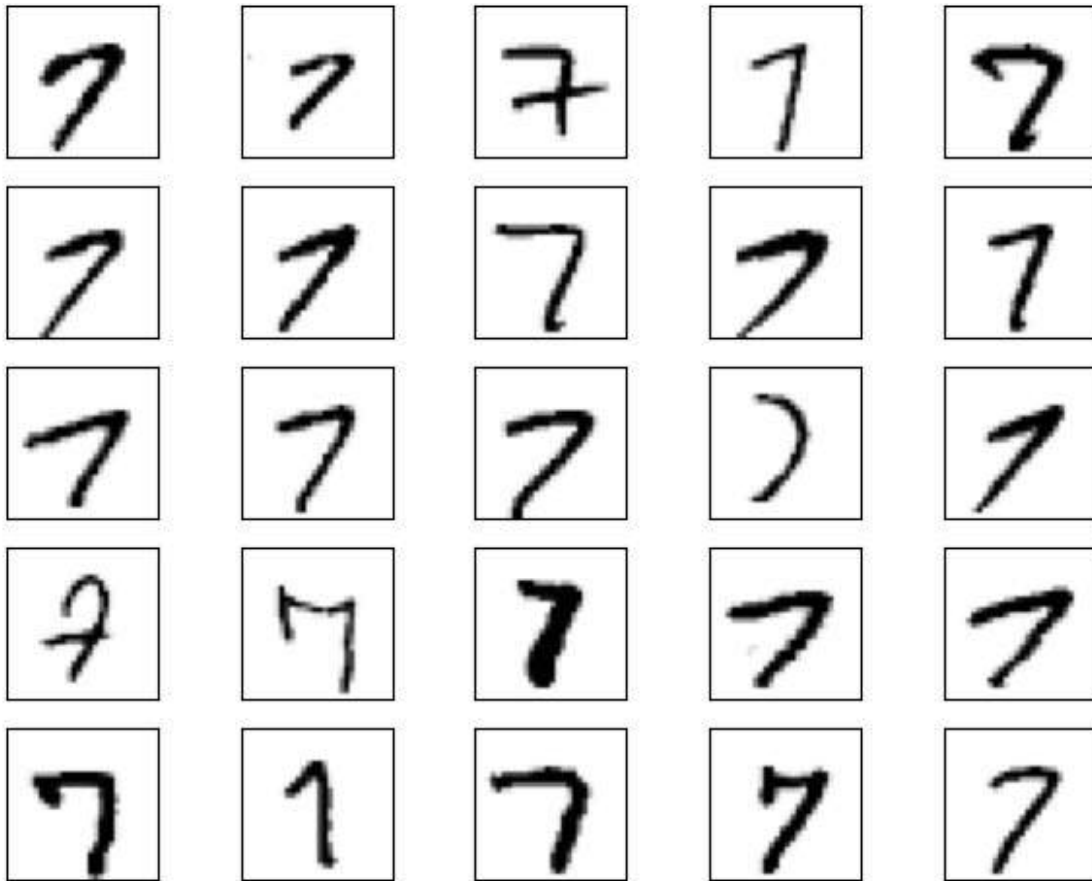
We should now see a plot of the 2×5 subfigures showing a representative image of each unique digit.

In addition, let's also plot multiple examples of the same digit to see how different the handwriting for each really is.

Visualize 25 different versions of "7":

```
fig, ax = plt.subplots(nrows=5, ncols=5, sharex=True, sharey=True,)
ax = ax.flatten()
for i in range(25):
    img = X_train[y_train == 7][i].reshape(28, 28)
    ax[i].imshow(img, cmap='Greys')

ax[0].set_xticks([])
ax[0].set_yticks([])
plt.tight_layout()
# plt.savefig('images/12_6.png', dpi=300)
plt.show()
```



After we've gone through all the previous steps, it is a good idea to save the scaled images in a format that we can load more quickly into a new Python session to avoid the overhead of reading in and processing the data again. When we are working with *NumPy* arrays, an efficient and convenient method to save multidimensional arrays to disk is *NumPy*'s `savez` function.

```
import numpy as np

np.savez_compressed('mnist_scaled.npz',
                    X_train=X_train,
                    y_train=y_train,
                    X_test=X_test,
                    y_test=y_test)
```

After we create the `.npz` files, we can load the preprocessed MNIST image arrays using *NumPy*'s `load` function as follows.

The `mnist` variable now refers to an object that can access the four data arrays that we provided as keyword arguments to the `savez_compressed` function. These input arrays are now listed under the `files` attribute `list` of the `mnist` object.

```
mnist = np.load('mnist_scaled.npz')
mnist.files

['X_train', 'y_train', 'X_test', 'y_test']
```

Using a list comprehension, we can retrieve all four data arrays as follows:

```
X_train, y_train, X_test, y_test = [mnist[f] for f in ['X_train',
                                                    'y_train',
                                                    'X_test', 'y_test']]

del mnist

X_train.shape
(60000, 784)
```

Implementing a multi-layer perceptron

We will now implement an MLP from scratch to classify the images in the MNIST dataset. To keep things simple, we will implement an MLP with only one hidden layer.

```
import numpy as np
import sys

class NeuralNetMLP(object):
    """ Feedforward neural network / Multi-layer perceptron
    classifier.

    Parameters
    -----
    n_hidden : int (default: 30)
        Number of hidden units.
    l2 : float (default: 0.)
        Lambda value for L2-regularization.
        No regularization if l2=0. (default)
    epochs : int (default: 100)
        Number of passes over the training set.
    eta : float (default: 0.001)
        Learning rate.
    shuffle : bool (default: True)
        Shuffles training data every epoch if True to prevent
        circles.
    minibatch_size : int (default: 1)
        Number of training examples per minibatch.
    seed : int (default: None)
        Random seed for initializing weights and shuffling.

    Attributes
    -----
    eval_ : dict
        Dictionary collecting the cost, training accuracy,
        and validation accuracy for each epoch during training.
    """
    def __init__(self, n_hidden=30,
                  l2=0., epochs=100, eta=0.001,
                  shuffle=True, minibatch_size=1, seed=None):
```

```

self.random = np.random.RandomState(seed)
self.n_hidden = n_hidden
self.l2 = l2
self.epochs = epochs
self.eta = eta
self.shuffle = shuffle
self.minibatch_size = minibatch_size

def _onehot(self, y, n_classes):
    """Encode labels into one-hot representation

    Parameters
    -----
    y : array, shape = [n_examples]
        Target values.
    n_classes : int
        Number of classes

    Returns
    -----
    onehot : array, shape = (n_examples, n_labels)

    """
    onehot = np.zeros((n_classes, y.shape[0]))
    for idx, val in enumerate(y.astype(int)):
        onehot[val, idx] = 1.
    return onehot.T

def _sigmoid(self, z):
    """Compute logistic function (sigmoid)"""
    return 1. / (1. + np.exp(-np.clip(z, -250, 250)))

def _forward(self, X):
    """Compute forward propagation step"""

    # step 1: net input of hidden layer
    # [n_examples, n_features] dot [n_features, n_hidden]
    # -> [n_examples, n_hidden]
    z_h = np.dot(X, self.w_h) + self.b_h

    # step 2: activation of hidden layer
    a_h = self._sigmoid(z_h)

    # step 3: net input of output layer
    # [n_examples, n_hidden] dot [n_hidden, n_classlabels]
    # -> [n_examples, n_classlabels]

    z_out = np.dot(a_h, self.w_out) + self.b_out

    # step 4: activation output layer
    a_out = self._sigmoid(z_out)

    return z_h, a_h, z_out, a_out

```



```

def _compute_cost(self, y_enc, output):
    """Compute cost function.

    Parameters
    -----
    y_enc : array, shape = (n_examples, n_labels)
        one-hot encoded class labels.
    output : array, shape = [n_examples, n_output_units]
        Activation of the output layer (forward propagation)

    Returns
    -----
    cost : float
        Regularized cost
    """
    L2_term = (self.l2 *
               (np.sum(self.w_h ** 2.) +
                np.sum(self.w_out ** 2.)))

    term1 = -y_enc * (np.log(output))
    term2 = (1. - y_enc) * np.log(1. - output)
    cost = np.sum(term1 - term2) + L2_term

    # If you are applying this cost function to other
    # datasets where activation
    # values maybe become more extreme (closer to zero or 1)
    # you may encounter "ZeroDivisionError"s due to numerical
    # instabilities in Python & NumPy for the current
implementation.
    # I.e., the code tries to evaluate log(0), which is
undefined.
    # To address this issue, you could add a small constant to
the
    # activation values that are passed to the log function.
    #
    # For example:
    #
    # term1 = -y_enc * (np.log(output + 1e-5))
    # term2 = (1. - y_enc) * np.log(1. - output + 1e-5)

    return cost

def predict(self, X):
    """Predict class labels

    Parameters
    -----
    X : array, shape = [n_examples, n_features]
        Input layer with original features.

    Returns:
    -----

```

```

        y_pred : array, shape = [n_examples]
            Predicted class labels.

        """
        z_h, a_h, z_out, a_out = self._forward(X)
        y_pred = np.argmax(z_out, axis=1)
        return y_pred

def fit(self, X_train, y_train, X_valid, y_valid):
    """ Learn weights from training data.

    Parameters
    -----
    X_train : array, shape = [n_examples, n_features]
        Input layer with original features.
    y_train : array, shape = [n_examples]
        Target class labels.
    X_valid : array, shape = [n_examples, n_features]
        Sample features for validation during training
    y_valid : array, shape = [n_examples]
        Sample labels for validation during training

    Returns:
    -----
    self

    """
    n_output = np.unique(y_train).shape[0] # number of class
labels
    n_features = X_train.shape[1]

    #####
    # Weight initialization
    #####

    # weights for input -> hidden
    self.b_h = np.zeros(self.n_hidden)
    self.w_h = self.random.normal(loc=0.0, scale=0.1,
self.n_hidden))
                                size=(n_features,

    # weights for hidden -> output
    self.b_out = np.zeros(n_output)
    self.w_out = self.random.normal(loc=0.0, scale=0.1,
n_output))
                                size=(self.n_hidden,

    epoch_strlen = len(str(self.epochs)) # for progress
formatting
    self.eval_ = {'cost': [], 'train_acc': [], 'valid_acc': []}

    y_train_enc = self._onehot(y_train, n_output)

    # iterate over training epochs

```

```

for i in range(self.epochs):

    # iterate over minibatches
    indices = np.arange(X_train.shape[0])

    if self.shuffle:
        self.random.shuffle(indices)

    for start_idx in range(0, indices.shape[0] -
self.minibatch_size +
                                1, self.minibatch_size):
        batch_idx = indices[start_idx:start_idx +
self.minibatch_size]

        # forward propagation
        z_h, a_h, z_out, a_out =
self._forward(X_train[batch_idx])

        #####
        # Backpropagation
        #####

        # [n_examples, n_classlabels]
        delta_out = a_out - y_train_enc[batch_idx]

        # [n_examples, n_hidden]
        sigmoid_derivative_h = a_h * (1. - a_h)

        # [n_examples, n_classlabels] dot [n_classlabels,
n_hidden]
        # -> [n_examples, n_hidden]
        delta_h = (np.dot(delta_out, self.w_out.T) *
                    sigmoid_derivative_h)

        # [n_features, n_examples] dot [n_examples,
n_hidden]
        # -> [n_features, n_hidden]
        grad_w_h = np.dot(X_train[batch_idx].T, delta_h)
        grad_b_h = np.sum(delta_h, axis=0)

        # [n_hidden, n_examples] dot [n_examples,
n_classlabels]
        # -> [n_hidden, n_classlabels]
        grad_w_out = np.dot(a_h.T, delta_out)
        grad_b_out = np.sum(delta_out, axis=0)

        # Regularization and weight updates
        delta_w_h = (grad_w_h + self.l2*self.w_h)
        delta_b_h = grad_b_h # bias is not regularized
        self.w_h -= self.eta * delta_w_h
        self.b_h -= self.eta * delta_b_h

        delta_w_out = (grad_w_out + self.l2*self.w_out)
        delta_b_out = grad_b_out # bias is not regularized

```

```

        self.w_out -= self.eta * delta_w_out
        self.b_out -= self.eta * delta_b_out

#####
# Evaluation
#####

# Evaluation after each epoch during training
z_h, a_h, z_out, a_out = self._forward(X_train)

cost = self._compute_cost(y_enc=y_train_enc,
                           output=a_out)

y_train_pred = self.predict(X_train)
y_valid_pred = self.predict(X_valid)

train_acc = ((np.sum(y_train ==
y_train_pred)).astype(np.float) /
              X_train.shape[0])
valid_acc = ((np.sum(y_valid ==
y_valid_pred)).astype(np.float) /
              X_valid.shape[0])

sys.stderr.write('\r%0*d/%d | Cost: %.2f '
                 '| Train/Valid Acc.: %.2f%%/%.2f%% ' %
                 (epoch_strlen, i+1, self.epochs, cost,
                  train_acc*100, valid_acc*100))
sys.stderr.flush()

self.eval_['cost'].append(cost)
self.eval_['train_acc'].append(train_acc)
self.eval_['valid_acc'].append(valid_acc)

return self

```

After executing this code, we next initialize a new 784 – 100 – 10 **MLP—an NN** with 784 input units (`n_features`), 100 hidden units (`n_hidden`), and 10 output units (`n_output`).

If you read through the `NeuralNetMLP` code, you've probably already guessed what these parameters are for. Here, you find a short summary of them:

- `l2` This is the λ parameter for $L2$ regularization to decrease the degree of overfitting.
- `epochs` This is the number of passes over the training dataset.
- `eta` This is the learning rate η .
- `shuffle` This is for shuffling the training set prior to every epoch to prevent the algorithm getting stuck in circles.
- `seed` This is a random seed for shuffling and weight initialization.
- `minibatch_size` This is the number of training examples in each minibatch when splitting the training data in each epoch for SGD. The gradient is computed for each mini-batch separately instead of the entire training data for faster learning.

Next, we train the MLP using 55,000 examples from the already shuffled MNIST training dataset and use the remaining 5,000 examples for validation during training. Note that training the NN may take up to five minutes on standard desktop computer hardware.

For instance, to load the training data into our current Python session, we will access the `X_train` array as follows (similar to a Python dictionary):

```
n_epochs = 200

nn = NeuralNetMLP(n_hidden=100,
                  l2=0.01,
                  epochs=n_epochs,
                  eta=0.0005,
                  minibatch_size=100,
                  shuffle=True,
                  seed=1)

nn.fit(X_train=X_train[:55000],
      y_train=y_train[:55000],
      X_valid=X_train[55000:],
      y_valid=y_train[55000:])

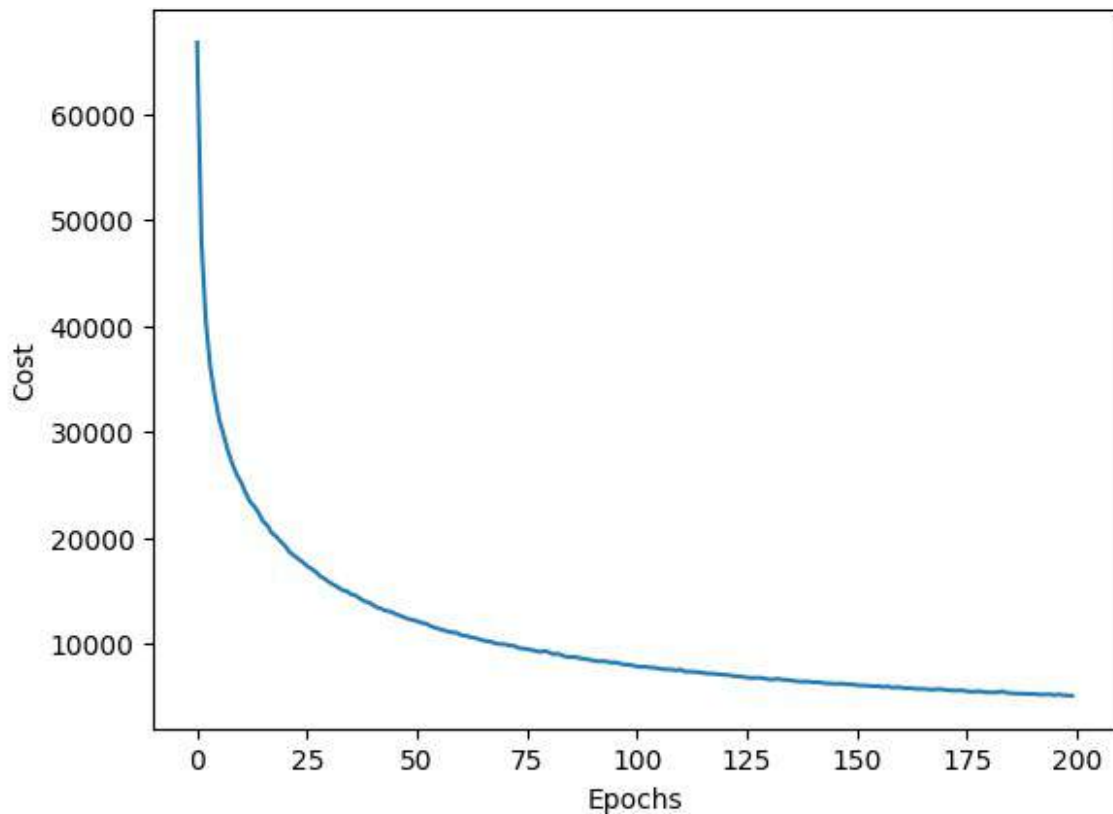
<ipython-input-35-d1456d3ec5e5>:255: DeprecationWarning: `np.float`
is a deprecated alias for the builtin `float`. To silence this
warning, use `float` by itself. Doing this will not modify any
behavior and is safe. If you specifically wanted the numpy scalar
type, use `np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance:
https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  train_acc = ((np.sum(y_train == y_train_pred)).astype(np.float) /
<ipython-input-35-d1456d3ec5e5>:257: DeprecationWarning: `np.float`
is a deprecated alias for the builtin `float`. To silence this
warning, use `float` by itself. Doing this will not modify any
behavior and is safe. If you specifically wanted the numpy scalar
type, use `np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance:
https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  valid_acc = ((np.sum(y_valid == y_valid_pred)).astype(np.float) /
200/200 | Cost: 5065.78 | Train/Valid Acc.: 99.28%/97.98%

<__main__.NeuralNetMLP at 0x78f0209461a0>
```

In our `NeuralNetMLP` implementation, we also defined an `eval_` attribute that collects the cost, training, and validation accuracy for each epoch so that we can visualize the results using **Matplotlib**.

```
import matplotlib.pyplot as plt

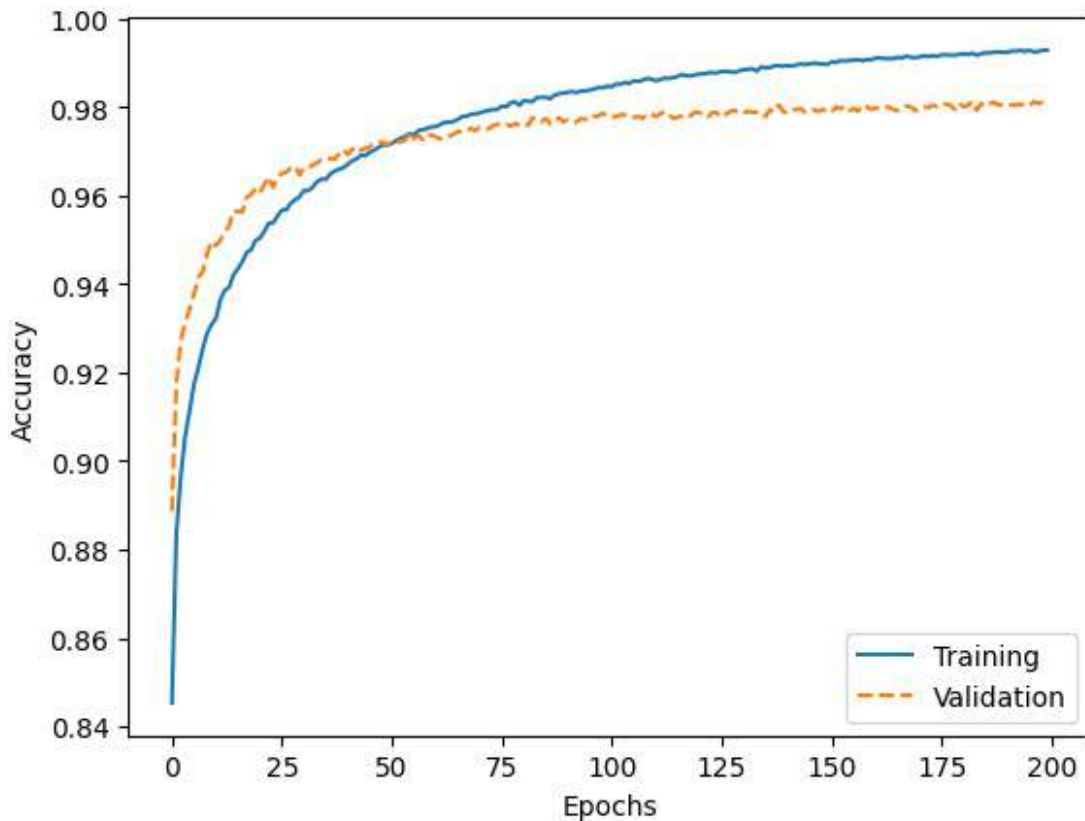
plt.plot(range(nn.epochs), nn.eval_['cost'])
plt.ylabel('Cost')
plt.xlabel('Epochs')
#plt.savefig('images/12_07.png', dpi=300)
plt.show()
```



As we can see, the cost decreased substantially during the first 100 epochs and seems to slowly converge in the last 100 epochs. However, the small slope between epoch 175 and epoch 200 indicates that the cost would further decrease with a training over additional epochs.

Next, let's take a look at the training and validation accuracy:

```
plt.plot(range(nn.epochs), nn.eval_['train_acc'],  
         label='Training')  
plt.plot(range(nn.epochs), nn.eval_['valid_acc'],  
         label='Validation', linestyle='--')  
plt.ylabel('Accuracy')  
plt.xlabel('Epochs')  
plt.legend(loc='lower right')  
plt.savefig('images/12_08.png', dpi=300)  
plt.show()
```



The plot reveals that the gap between training and validation accuracy increases as we train for more epochs. At approximately the 50th epoch, the training and validation accuracy values are equal, and then, the network starts overfitting the training data.

Note that this example was chosen deliberately to illustrate the effect of overfitting and demonstrate why it is useful to compare the validation and training accuracy values during training. One way to decrease the effect of overfitting is to increase the regularization strength—for example, by setting $\lambda_2 = 0.1$. Another useful technique to tackle overfitting in NNs is dropout, which will be covered in the next part of our session (*Classifying Images with Deep Convolutional Neural Networks*).

Finally, let's evaluate the generalization performance of the model by calculating the prediction accuracy on the test dataset:

```
y_test_pred = nn.predict(X_test)
acc = (np.sum(y_test == y_test_pred)
      .astype(float) / X_test.shape[0])

print('Test accuracy: %.2f%%' % (acc * 100))

Test accuracy: 97.54%
```

Despite the slight overfitting on the training data, our relatively simple one-hiddenlayer NN achieved a relatively good performance on the test dataset, similar to the validation dataset accuracy (97.98 percent).

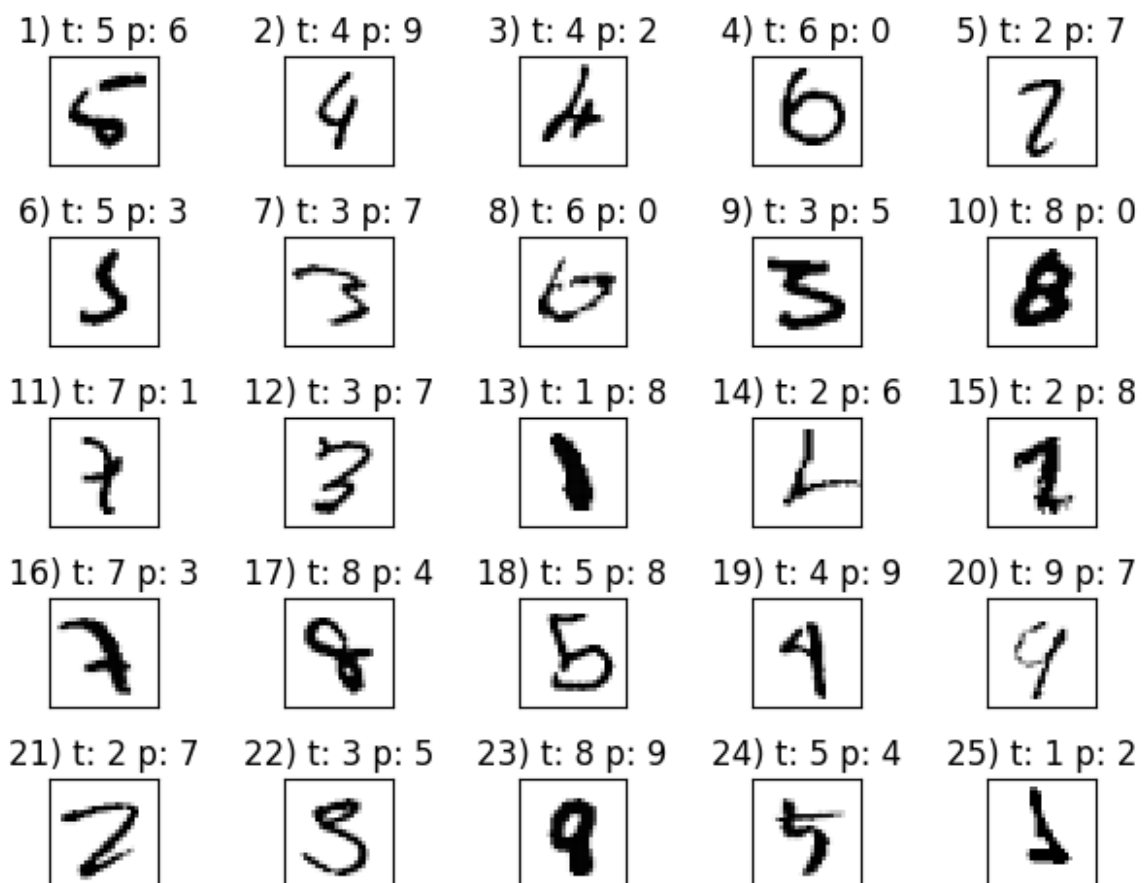
To further fine-tune the model, we could change the number of hidden units, values of the regularization parameters, and the learning rate, or use various other tricks that have been developed over the years.

Lastly, let's take a look at some of the images that our MLP struggles with:

```
misc_img = X_test[y_test != y_test_pred][:25]
correct_lab = y_test[y_test != y_test_pred][:25]
misc_lab = y_test_pred[y_test != y_test_pred][:25]

fig, ax = plt.subplots(nrows=5, ncols=5, sharex=True, sharey=True)
ax = ax.flatten()
for i in range(25):
    img = misc_img[i].reshape(28, 28)
    ax[i].imshow(img, cmap='Greys', interpolation='nearest')
    ax[i].set_title('%d) t: %d p: %d' % (i+1, correct_lab[i],
misc_lab[i]))

ax[0].set_xticks([])
ax[0].set_yticks([])
plt.tight_layout()
#plt.savefig('images/12_09.png', dpi=300)
plt.show()
```



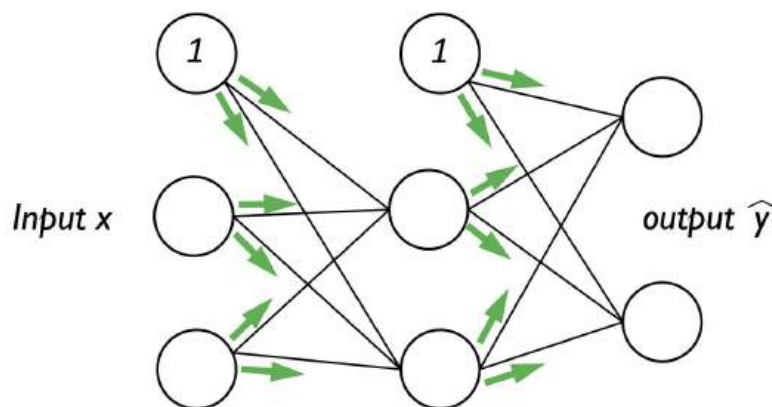
As we can see in the preceding figure, some of those images are even challenging for us humans to classify correctly. For example, the 6 in subplot 8 really looks like a carelessly drawn 0, and the 8 in subplot 23 could be a 9 due to the narrow lower part combined with the bold line.

2.4. Training neural networks via backpropagation

In this section, we will go through the math of backpropagation to understand how you can learn the weights in an NN very efficiently. In a previous section, we saw how to calculate the *cost* as the difference between the activation of the last layer and the target class label. Now, we will see how the backpropagation algorithm works to update the weights in our MLP model from a mathematical perspective, which we implemented after the `# Backpropagation` code comment inside the `fit` method.

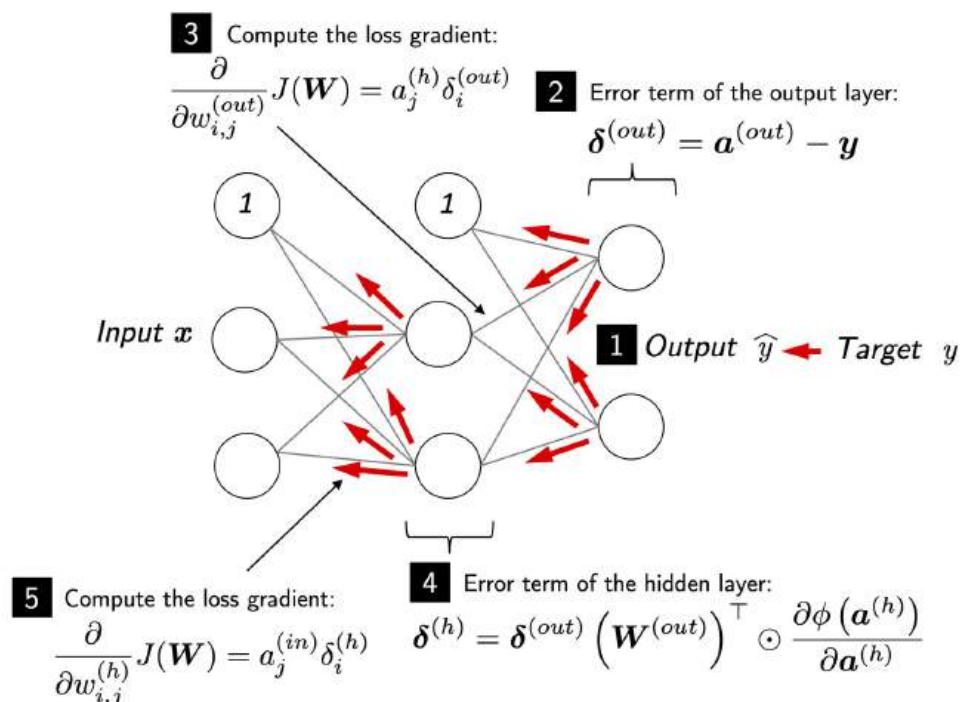
Concisely, we just forward-propagate the input features through the connection in the network, as shown in the following illustration:

```
Image(filename='./images/12_11.png', width=400)
```



Mathematically we can summarize backpropagation in the following figure:

```
Image(filename='images/12_12.png', width=500)
```



3. Convolutional Neural Networks (CNNs)

- a. Introduction to Convolutional Neural Networks
- b. Convolution and pooling layers
- c. Transfer learning and pre-trained CNNs
- d. Applications of CNNs in image recognition and computer vision

3.1. The building blocks of CNNs

CNNs are a family of models that were originally inspired by how the visual cortex of the human brain works when recognizing objects. The development of CNNs goes back to the 1990s, when Yann LeCun and his colleagues proposed a novel NN architecture for classifying handwritten digits from images. Due to the outstanding performance of CNNs for image classification tasks, this particular type of feedforward NN gained a lot of attention and led to tremendous improvements in machine learning for computer vision.

Successfully extracting **salient** (relevant) features is key to the performance of any machine learning algorithm and traditional machine learning models rely on input features that may come from a domain expert or are based on computational feature extraction techniques.

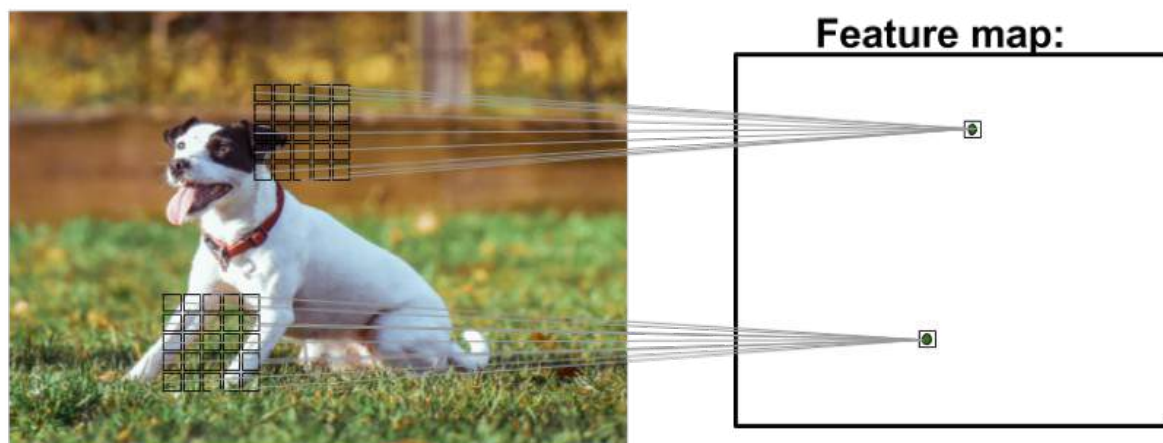
Certain types of NNs, such as CNNs, are able to automatically learn the features from raw data that are most useful for a particular task. For this reason, it's common to consider CNN layers as feature extractors: the early layers (those right after the input layer) extract low-level features from raw data, and the later layers (often fully connected layers like in a multilayer perceptron (MLP)) use these features to predict a continuous target value or class label.

Certain types of multilayer NNs, and in particular, deep convolutional NNs, construct a so-called feature hierarchy by combining the low-level features in a layer-wise fashion to form high-level features. For example, if we're dealing with images, then low-level features, such as edges and blobs, are extracted from the earlier layers, which are combined together to form high-level features. These highlevel features can form more complex shapes, such as the general contours of objects like buildings, cats, or dogs.

As you can see in the following image, a CNN computes feature maps from an input image, where each element comes from a local patch of pixels in the input image:

```
from IPython.display import Image
%matplotlib inline

Image(filename='images/15_01.png', width=700)
```



This local patch of pixels is referred to as the local receptive field. CNNs will usually perform very well on image-related tasks, and that's largely due to two important ideas:

- **Sparse connectivity:** A single element in the feature map is connected to only a small patch of pixels. (This is very different from connecting to the whole input image as in the case of perceptrons. You may find it useful to look back and compare how we implemented a fully connected network that connected to the whole image in Chapter 12, Implementing a Multilayer Artificial Neural Network from Scratch.)
- **Parameter-sharing:** The same weights are used for different patches of the input image.

As a direct consequence of these two ideas, replacing a conventional, fully connected MLP with a convolution layer substantially decreases the number of weights (parameters) in the network and we will see an improvement in the ability to capture salient features. In the context of image data, it makes sense to assume that nearby pixels are typically more relevant to each other than pixels that are far away from each other.

Typically, CNNs are composed of several convolutional and subsampling layers that are followed by one or more fully connected layers at the end. The fully connected layers are essentially an MLP, where every input unit, i , is connected to every output unit, j , with weight w_{ij} . Please note that subsampling layers, commonly known as **pooling layers**, do not have any learnable parameters; for instance, there are no weights or bias units in pooling layers. However, both the convolutional and fully connected layers have weights and biases that are optimized during training.

Performing discrete convolutions

A discrete convolution (or simply convolution) is a fundamental operation in a CNN. Therefore, it's important to understand how this operation works. In this section, we will cover the mathematical definition and discuss some of the naive algorithms to compute convolutions of one-dimensional tensors (vectors) and two-dimensional tensors (matrices).

Discrete convolutions in one dimension

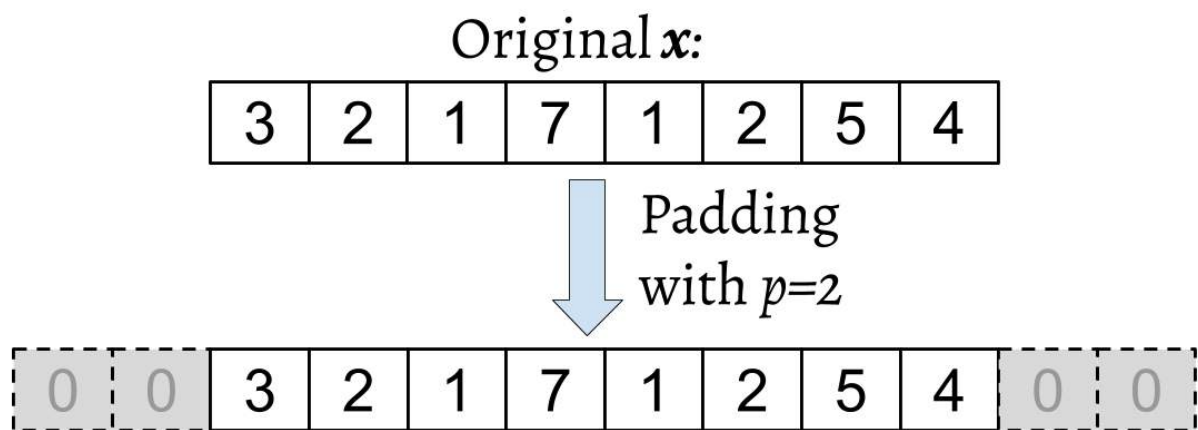
A discrete convolution for two vectors, x and w , is denoted by $y = x * w$, in which vector x is our input (sometimes called *signal*) and w is called the *filter* or *kernel*. A discrete

convolution is mathematically defined as follows: $y = x * w \rightarrow \sum_{k=-\infty}^{+\infty} x[i-k] * w[k]$

The brackets, $[\]$, are used to denote the indexing for vector elements. The index, i , runs through each element of the output vector, y . There are two odd things in the preceding formula that we need to clarify: $-\infty$ to $+\infty$ indices and negative indexing for x . The fact that the sum runs through indices from $-\infty$ to $+\infty$ seems odd, mainly because in machine learning applications, we always deal with finite feature vectors. For example, if x has 10 features with indices $0, 1, 2, \dots, 8, 9$, then indices $-\infty: -1$ and $10: +\infty$ are out of bounds for x . Therefore, to correctly compute the summation shown in the preceding formula, it is assumed that x and w are filled with zeros. This will result in an output vector, y , that also has infinite size, with lots of zeros as well. Since this is not useful in practical situations, x is padded only with a finite number of zeros.

This process is called **zero-padding** or simply **padding**. Here, the number of zeros padded on each side is denoted by p . An example padding of a one-dimensional vector, x , is shown in the following figure:

```
Image(filename='images/15_02.png', width=700)
```



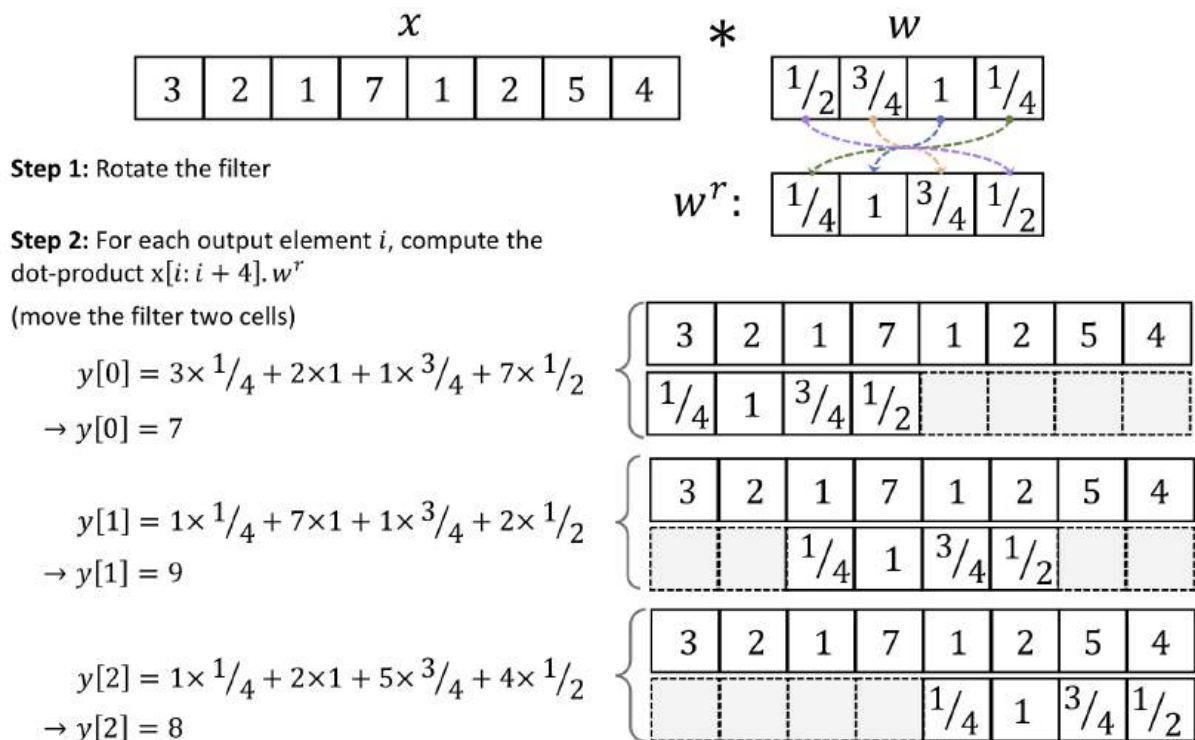
Let's assume that the original input, x , and filter, w , have n and m elements, respectively, where $m \leq n$. Therefore, the padded vector, x^p , has size $n+2p$. The practical formula for computing a discrete convolution will change to the following:

$$y = x * w \rightarrow y[i] = \sum_{k=0}^{m-1} x^p[i+m-k] * w[k]$$

Now that we have solved the infinite index issue, the second issue is indexing x with $i+m-k$. The important point to notice here is that x and w are indexed in different directions in this summation. Computing the sum with one index going in the reverse direction is equivalent to computing the sum with both indices in the forward direction after flipping one of those vectors, x or w , after they are padded.

Then, we can simply compute their dot product. Let's assume we flip (rotate) the filter, w , to get the rotated filter, w^r . Then, the dot product, $x[i:i+m] \cdot w^r$, is computed to get one element, $y[i]$, where $x[i:i+m]$ is a patch of x with size m . This operation is repeated like in a sliding window approach to get all the output elements. The following figure provides an example with $x = [3 \text{ } 2 \text{ } 1 \text{ } 7 \text{ } 1 \text{ } 2 \text{ } 5 \text{ } 4]$ and $w = [\frac{1}{2} \text{ } \frac{3}{4} \text{ } 1 \text{ } \frac{1}{4}]$ so that the first three output elements are computed:

```
Image(filename='images/15_03.png', width=700)
```



Padding inputs to control the size of the output feature maps

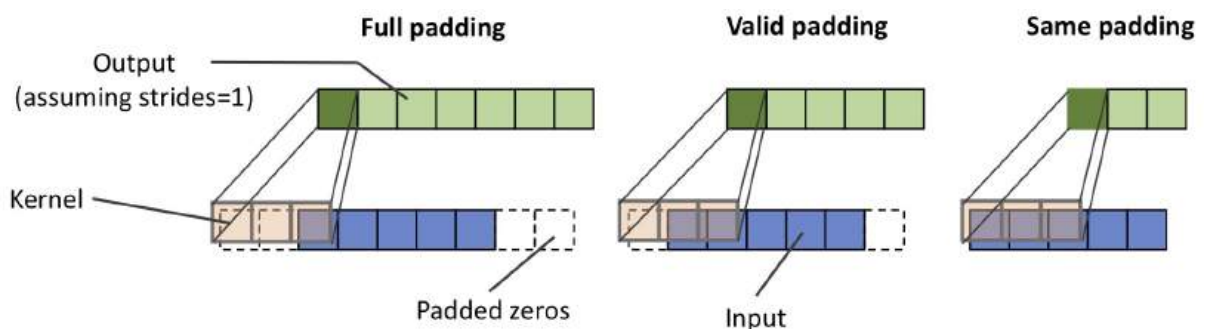
The size of the output, y , also depends on the choice of the padding strategy we use.

There are three modes of padding that are commonly used in practice: *full*, *same*, and *valid*:

- In **full** mode, the padding parameter, p , is set to $p=m-1$. Full padding increases the dimensions of the output; thus, it is rarely used in CNN architectures.
- **Same** padding is usually used to ensure that the output vector has the same size as the input vector, x . In this case, the padding parameter, p , is computed according to the filter size, along with the requirement that the input size and output size are the same.
- Finally, computing a convolution in the **valid** mode refers to the case where $p=0$ (no padding).

The following figure illustrates the three different padding modes for a simple 5×5 pixel input with a kernel size of 3×3 and a stride of 1:

```
Image(filename='images/15_04.png', width=700)
```



In practice, it is recommended that you preserve the spatial size using same padding for the convolutional layers and decrease the spatial size via pooling layers instead. As for full

padding, its size results in an output larger than the input size. Full padding is usually used in signal processing applications where it is important to minimize boundary effects. However, in the deep learning context, boundary effects are usually not an issue, so we rarely see full padding being used in practice.

Determining the size of the convolution output

The output size of a convolution is determined by the total number of times that we shift the filter, w , along the input vector. Let's assume that the input vector is of size n and the filter is of size m . Then, the size of the output resulting from $\mathbf{y} = \mathbf{x} * \mathbf{w}$, with padding, p , and stride, s , would be determined as follows: $o = \lfloor \frac{n+2p-m}{s} \rfloor + 1$. Here, $\lfloor \cdot \rfloor$ denotes the *floor* operation.

Finally, in order to learn how to compute convolutions in one dimension, a naive implementation is shown in the following code block, and the results are compared with the `numpy.convolve` function. The code is as follows:

```
import tensorflow as tf
import numpy as np

print('TensorFlow version:', tf.__version__)
print('NumPy version: ', np.__version__)

def conv1d(x, w, p=0, s=1):
    w_rot = np.array(w[::-1])
    x_padded = np.array(x)
    if p > 0:
        zero_pad = np.zeros(shape=p)
        x_padded = np.concatenate(
            [zero_pad, x_padded, zero_pad])
    res = []
    for i in range(0, int((len(x_padded) - len(w_rot)) / s) + 1, s):
        res.append(np.sum(
            x_padded[i:i+w_rot.shape[0]] * w_rot))
    return np.array(res)

## Testing:
x = [1, 3, 2, 4, 5, 6, 1, 3]
w = [1, 0, 3, 1, 2]

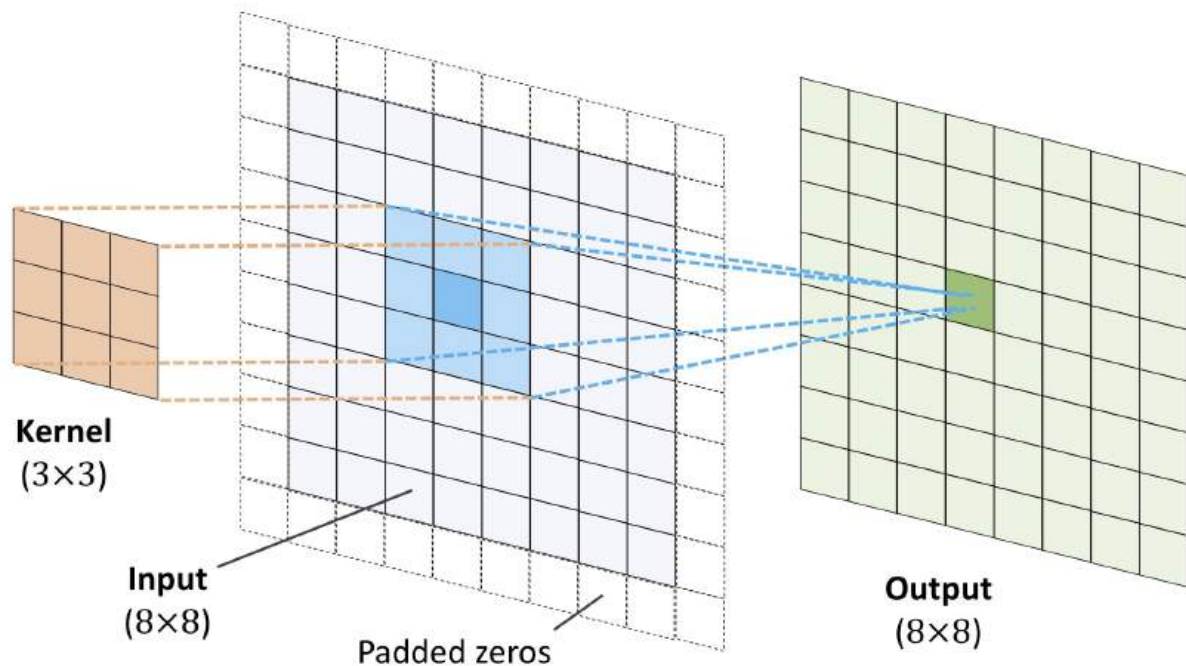
print('Conv1d Implementation:',
      conv1d(x, w, p=2, s=1))

print('Numpy Results:',
      np.convolve(x, w, mode='same'))
```

Performing a discrete convolution in 2D

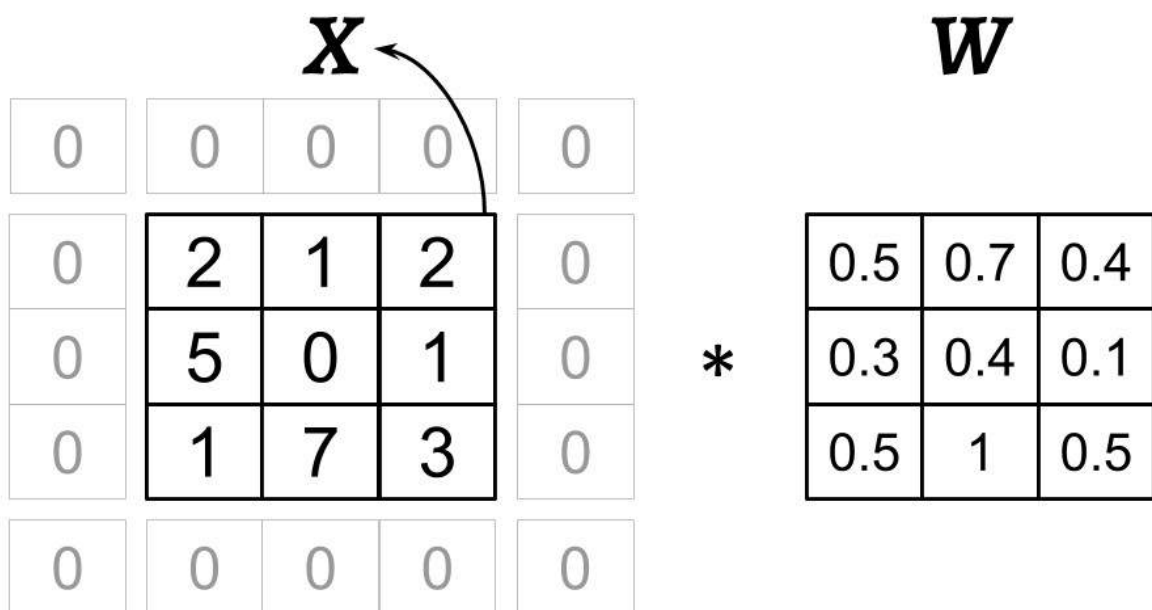
The concepts you learned in the previous sections are easily extendible to 2D. The following figure demonstrates **2D** convolution of an input matrix of size 8×8 , using a kernel of size 3×3 . The input matrix is padded with zeros with $p=1$. As a result, the output of the 2D convolution will have a size of 8×8 :


```
Image(filename='images/15_05.png', width=700)
```



The following example illustrates the computation of a 2D convolution between an input matrix, $\mathbf{X}_{3 \times 3}$, and a kernel matrix, $\mathbf{W}_{3 \times 3}$, using padding $p=(1, 1)$ and stride $s=(2, 2)$. According to the specified padding, one layer of zeros is added on each side of the input matrix, which results in the padded matrix $\mathbf{X}_{5 \times 5}$ padded, as follows:

```
Image(filename='images/15_06.png', width=600)
```

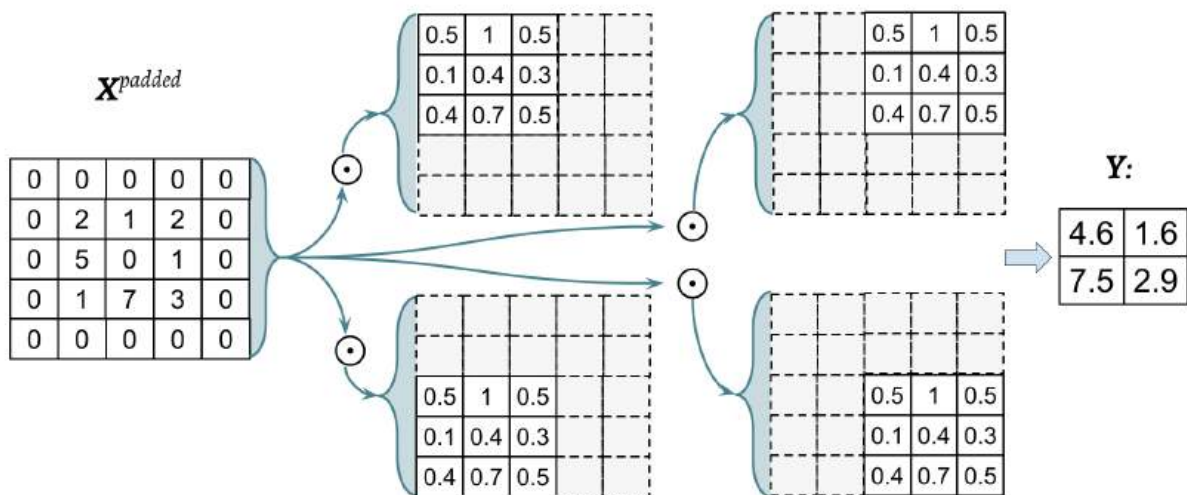


With the preceding filter, the rotated filter will be:

$W_r = \begin{bmatrix} 0.5 & 1 & 0.5 & 0.1 & 0.4 & 0.3 \\ 0.4 & 0.7 & 0.5 \end{bmatrix}$

Note that this rotation is not the same as the transpose matrix. To get the rotated filter in NumPy, we can write `W_rot=W[::-1,:-1]`. Next, we can shift the rotated filter matrix along the padded input matrix, X^{padded} , like a sliding window and compute the sum of the element-wise product, which is denoted by the $\odot$ operator in the following figure:

Image(filename='images/15_07.png', width=800)



The result will be the 2×2 matrix, Y . Let's also implement the 2D convolution according to the naive algorithm described. The `scipy.signal` package provides a way to compute 2D convolution via the `scipy.signal.convolve2d` function:

```
import scipy.signal

def conv2d(X, W, p=(0, 0), s=(1, 1)):
    W_rot = np.array(W)[::-1,::-1]
    X_orig = np.array(X)
    n1 = X_orig.shape[0] + 2*p[0]
    n2 = X_orig.shape[1] + 2*p[1]
    X_padded = np.zeros(shape=(n1, n2))
    X_padded[p[0]:p[0]+X_orig.shape[0],
             p[1]:p[1]+X_orig.shape[1]] = X_orig

    res = []
    for i in range(0, int((X_padded.shape[0] -
                           W_rot.shape[0])/s[0])+1, s[0]):
        res.append([])
        for j in range(0, int((X_padded.shape[1] -
                               W_rot.shape[1])/s[1])+1, s[1]):
            X_sub = X_padded[i:i+W_rot.shape[0],
                              j:j+W_rot.shape[1]]
            res[-1].append(np.sum(X_sub * W_rot))
    return np.array(res)
```



```
X = [[1, 3, 2, 4], [5, 6, 1, 3], [1, 2, 0, 2], [3, 4, 3, 2]]
W = [[1, 0, 3], [1, 2, 1], [0, 1, 1]]
```

```
print('Conv2d Implementation:\n',
      conv2d(X, W, p=(1, 1), s=(1, 1)))
```

```
print('SciPy Results:\n',
      scipy.signal.convolve2d(X, W, mode='same'))
```

Conv2d Implementation:

```
[[11. 25. 32. 13.]
 [19. 25. 24. 13.]
 [13. 28. 25. 17.]
 [11. 17. 14.  9.]]
```

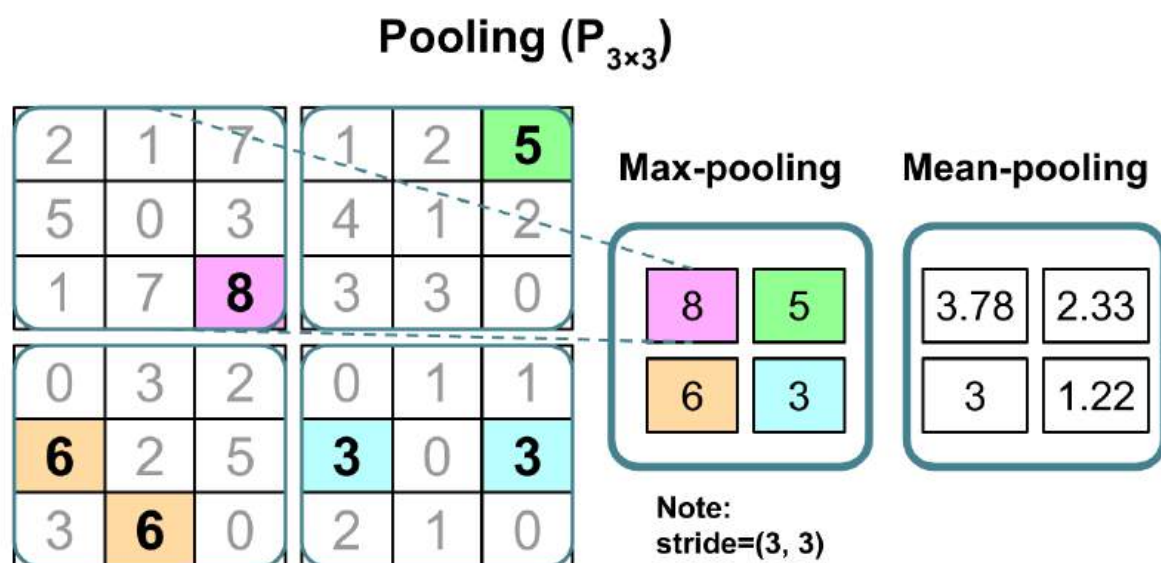
SciPy Results:

```
[[11 25 32 13]
 [19 25 24 13]
 [13 28 25 17]
 [11 17 14  9]]
```

Subsampling layers

Subsampling is typically applied in two forms of *pooling* operations in CNNs: **max-pooling** and **mean-pooling** (also known as **average-pooling**). The pooling layer is usually denoted by $P_{n_1 \times n_2}$. Here, the subscript determines the size of the neighborhood (the number of adjacent pixels in each dimension) where the max or mean operation is performed. We refer to such a neighborhood as the pooling size. The operation is described in the following figure. Here, max-pooling takes the maximum value from a neighborhood of pixels, and mean-pooling computes their average:

```
Image(filename='images/15_08.png', width=700)
```



The advantage of pooling is twofold:

- Pooling (max-pooling) introduces a local invariance. This means that small changes in a local neighborhood do not change the result of max-pooling.
- Pooling decreases the size of features, which results in higher computational efficiency. Furthermore, reducing the number of features may reduce the degree of overfitting as well.

Putting everything together – implementing a CNN

In a CNN, the operation matrix multiplication is replaced by a convolution operation, as in $\mathbf{Z} = \mathbf{W} * \mathbf{X} + \mathbf{b}$, where $\mathbf{X}$ is a matrix representing the pixels in a *height* $\times$ *weight* arrangement. In both cases (NN and CNN), the pre-activations are passed to an activation function to obtain the activation of a hidden unit, $\mathbf{A} = \phi(\mathbf{Z})$, where ϕ is the activation function. Furthermore, you will recall that subsampling is another building block of a CNN, which may appear in the form of pooling, as was described in the previous section.

Working with multiple input or color channels

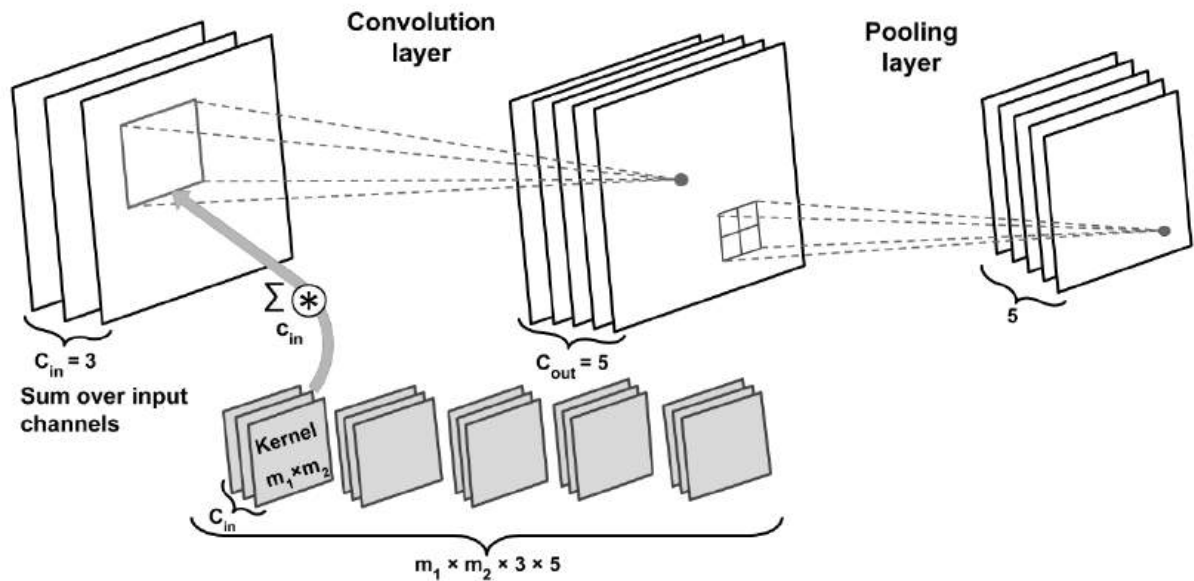
An input to a convolutional layer may contain one or more 2D arrays or matrices with dimensions $N_1 \times N_2$ (for example, the image height and width in pixels). These $N_1 \times N_2$ matrices are called channels. Conventional implementations of convolutional layers expect a *rank-3* tensor representation as an input, for example a three-dimensional array, $\mathbf{X}_{N_1 \times N_2 \times C_{in}}$ where C_{in} is the number of input channels. For example, let's consider images as input to the first layer of a CNN. If the image is colored and uses the **RGB** color mode, then $C_{in} = 3$ (for the red, green, and blue color channels in RGB). However, if the image is in grayscale, then we have $C_{in} = 1$, because there is only one channel with the grayscale pixel intensity values.

Now that you are familiar with the structure of input data, the next question is, how can we incorporate multiple input channels in the convolution operation that we discussed in the previous sections? The answer is very simple: we perform the convolution operation for each channel separately and then add the results together using the matrix summation. The convolution associated with each channel (c) has its own kernel matrix as $W[:, :, c]$.

The final result is a feature map. Usually, a convolutional layer of a CNN has more than one feature map. If we use multiple feature maps, the kernel tensor becomes four-dimensional: *weight* $\times$ *height* $\times$ $C_{in} \times C_{out}$. Here, *weight* $\times$ *height* is the kernel size, C_{in} is the number of input channels, and C_{out} is the number of output feature maps.

To conclude our discussion of computing convolutions in the context of NNs, let's look at the example in the following figure, which shows a convolutional layer, followed by a pooling layer. In this example, there are three input channels. The kernel tensor is four-dimensional. Each kernel matrix is denoted as $m_1 \times m_2$, and there are three of them, one for each input channel. Furthermore, there are five such kernels, accounting for five output feature maps. Finally, there is a pooling layer for subsampling the feature maps:

```
Image(filename='images/15_09.png', width=800)
```



Lastly, as was already mentioned, typically, the convolution operations are carried out by treating an input image with multiple color channels as a stack of matrices; that is, we perform the convolution on each matrix separately and then add the results, as was illustrated in the previous figure. However, convolutions can also be extended to 3D volumes if you are working with 3D datasets.

TIP: Reading an image file

```
import tensorflow as tf

img_raw = tf.io.read_file('example-image.png')
img = tf.image.decode_image(img_raw)
print('Image shape:', img.shape)
print('Number of channels:', img.shape[2])
print('Image data type:', img.dtype)
print(img[100:102, 100:102, :])

Image shape: (252, 221, 3)
Number of channels: 3
Image data type: <dtype: 'uint8'>
tf.Tensor(
[[[179 134 110]
  [182 136 112]]

 [[180 135 111]
  [182 137 113]]], shape=(2, 2, 3), dtype=uint8)

import imageio

img = imageio.imread('example-image.png')
print('Image shape:', img.shape)
print('Number of channels:', img.shape[2])
print('Image data type:', img.dtype)
print(img[100:102, 100:102, :])
```

```
Image shape: (252, 221, 3)
Number of channels: 3
Image data type: uint8
[[[179 134 110]
  [182 136 112]]

 [[180 135 111]
  [182 137 113]]]
```

INFO-BOX: The rank of a grayscale image for input to a CNN

```
img_raw = tf.io.read_file('example-image-gray.png')
img = tf.image.decode_image(img_raw)
tf.print('Rank:', tf.rank(img))
tf.print('Shape:', img.shape)

Rank: 3
Shape: TensorShape([252, 221, 1])

img = imageio.imread('example-image-gray.png')
tf.print('Rank:', tf.rank(img))
tf.print('Shape:', img.shape)

img_resaped = tf.reshape(img, (img.shape[0], img.shape[1], 1))
tf.print('New Shape:', img_resaped.shape)

Rank: 2
Shape: (252, 221)
New Shape: TensorShape([252, 221, 1])
```

Regularizing a neural network with dropout

Choosing the size of a network, whether we are dealing with a traditional (fully connected) NN or a CNN, has always been a challenging problem. For instance, the size of a weight matrix and the number of layers need to be tuned to achieve a reasonably good performance.

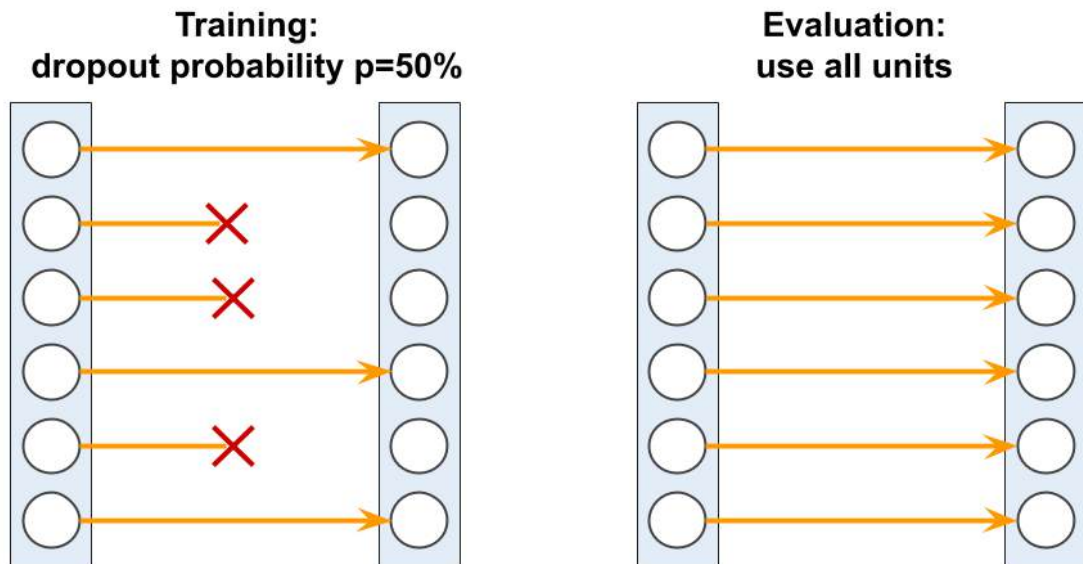
In recent years, **dropout** has emerged as a popular technique for regularizing (deep) NNs to avoid overfitting, thus improving the generalization performance. Dropout is usually applied to the hidden units of higher layers and works as follows: during the training phase of an NN, a fraction of the hidden units is randomly dropped at every iteration with probability p_{drop} (or keep probability $p_{keep}=1-p_{drop}$). This dropout probability is determined by the user and the common choice is $p=0.5$. When dropping a certain fraction of input neurons, the weights associated with the remaining neurons are rescaled to account for the missing (dropped) neurons.

The effect of this random dropout is that the network is forced to learn a redundant representation of the data. Therefore, the network cannot rely on an activation of any set of hidden units, since they may be turned off at any time during training, and is forced to learn more general and robust patterns from the data.

This random dropout can effectively prevent overfitting. The following figure shows an example of applying dropout with probability $p=0.5$ during the training phase, whereby half of the neurons will become inactive randomly (dropped units are selected randomly in each

forward pass of training). However, during prediction, all neurons will contribute to computing the pre-activations of the next layer.

```
Image(filename='images/15_10.png', width=700)
```



```
from tensorflow import keras

conv_layer = keras.layers.Conv2D(
    filters=16, kernel_size=(3, 3),
    kernel_regularizer=keras.regularizers.l2(0.001))

fc_layer = keras.layers.Dense(
    units=16, kernel_regularizer=keras.regularizers.l2(0.001))
```

Loss Functions for Classification

Focusing on classification problems here, depending on the type of problem (binary versus multiclass) and the type of output (logits versus probabilities), we should choose the appropriate loss function to train our model. Binary cross-entropy is the loss function for a binary classification (with a single output unit), and categorical cross-entropy is the loss function for multiclass classification. In the Keras API, two options for categorical cross-entropy loss are provided, depending on whether the ground truth labels are in a one-hot encoded format (for example, $[0, 0, 1, 0]$), or provided as integer labels (for example, $y = 2$), which is also known as "sparse" representation in the context of Keras.

- **BinaryCrossentropy()**
 - from_logits=False
 - from_logits=True
- **CategoricalCrossentropy()**
 - from_logits=False
 - from_logits=True
- **SparseCategoricalCrossentropy()**

- `from_logits=False`
- `from_logits=True`

The following table describes three loss functions available in Keras for dealing with all three cases: binary classification, multiclass with one-hot encoded ground truth labels, and multiclass with integer (sparse) labels. Each one of these three loss functions also has the option to receive the predictions in the form of logits or classmembership probabilities:

```
Image(filename='images/15_11.png', width=800)
```

Loss function	Usage	Examples	
		Using probabilities	Using logits
		<code>from_logits=False</code>	<code>from_logits=True</code>
BinaryCrossentropy	Binary classification	y_true: 1 y_pred: 0.69	y_true: 1 y_pred: 0.8
CategoricalCrossentropy	Multiclass classification	y_true: 0 0 1 y_pred: 0.30 0.15 0.55	y_true: 0 0 1 y_pred: 1.5 0.8 2.1
Sparse CategoricalCrossentropy	Multiclass classification	y_true: 2 y_pred: 0.30 0.15 0.55	y_true: 2 y_pred: 1.5 0.8 2.1

The following code will show you how to use these three loss functions with two different formats, where either the logits or class-membership probabilities are given as inputs to the loss functions:

```
from distutils.version import LooseVersion as Version

##### Binary Crossentropy
bce_probas = tf.keras.losses.BinaryCrossentropy(from_logits=False)
bce_logits = tf.keras.losses.BinaryCrossentropy(from_logits=True)

logits = tf.constant([0.8])
probas = tf.keras.activations.sigmoid(logits)

if Version(tf.__version__) >= '2.3.0':
    tf.print(
        'CCE (w Probas): {:.4f}'.format(
            cce_probas(y_true=[[0, 0, 1]], y_pred=probas)),
        '(w Logits): {:.4f}'.format(
            cce_logits(y_true=[[0, 0, 1]], y_pred=logits)))
else:
    tf.print(
        'CCE (w Probas): {:.4f}'.format(
            cce_probas(y_true=[0, 0, 1], y_pred=probas)),
        '(w Logits): {:.4f}'.format(
            cce_logits(y_true=[0, 0, 1], y_pred=logits)))
```

```
##### Categorical Crossentropy
cce_probas = tf.keras.losses.CategoricalCrossentropy(
    from_logits=False)
cce_logits = tf.keras.losses.CategoricalCrossentropy(
    from_logits=True)

logits = tf.constant([[1.5, 0.8, 2.1]])
probas = tf.keras.activations.softmax(logits)

tf.print(
    'CCE (w Probas): {:.4f}'.format(
        cce_probas(y_true=[0, 0, 1], y_pred=probas)),
    '(w Logits): {:.4f}'.format(
        cce_logits(y_true=[0, 0, 1], y_pred=logits)))

##### Sparse Categorical Crossentropy
sp_cce_probas = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=False)
sp_cce_logits = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True)

tf.print(
    'Sparse CCE (w Probas): {:.4f}'.format(
        sp_cce_probas(y_true=[2], y_pred=probas)),
    '(w Logits): {:.4f}'.format(
        sp_cce_logits(y_true=[2], y_pred=logits)))

BCE (w Probas): 0.3711 (w Logits): 0.3711
CCE (w Probas): 0.5996 (w Logits): 0.5996
Sparse CCE (w Probas): 0.5996 (w Logits): 0.5996
```

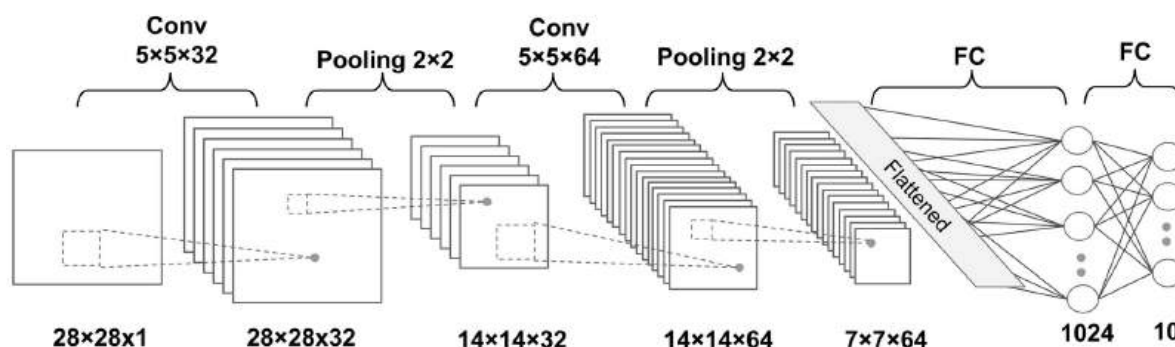
Implementing a deep convolutional neural network using TensorFlow

Now, let's implement a CNN and see whether it can achieve a better predictive performance compared to the MLP (DNNClassifier) for classifying handwritten digits. However, in some applications, such as reading bank account numbers from handwritten digits, even tiny mistakes can be very costly. Therefore, it is crucial to reduce this error as much as possible.

The multilayer CNN architecture

The architecture of the network that we are going to implement is shown in the following figure. The inputs are 28×28 grayscale images. Considering the number of channels (which is 1 for grayscale images) and a batch of input images, the input tensor's dimensions will be $batch\ size \times 28 \times 28 \times 1$. The input data goes through two convolutional layers that have a kernel size of 5×5 . The first convolution has **32** output feature maps, and the second one has **64** output feature maps. Each convolution layer is followed by a subsampling layer in the form of a max-pooling operation, $P_{2 \times 2}$. Then a fully connected layer passes the output to a second fully connected layer, which acts as the final softmax output layer. The architecture of the network that we are going to implement is shown in the following figure:


```
Image(filename='images/15_12.png', width=800)
```



The dimensions of the tensors in each layer are as follows:

- Input: $[batchsize \times 28 \times 28 \times 1]$
- Conv_1: $[batchsize \times 28 \times 28 \times 32]$
- Pooling_1: $[batchsize \times 14 \times 14 \times 32]$
- Conv_2: $[batchsize \times 14 \times 14 \times 64]$
- Pooling_2: $[batchsize \times 7 \times 7 \times 64]$
- FC_1: $[batchsize \times 1024]$
- FC_2 and softmax layer: $[batchsize \times 10]$

For the convolutional kernels, we are using $stride=1$ such that the input dimensions are preserved in the resulting feature maps. For the pooling layers, we are using $stride=2$ to subsample the image and shrink the size of the output feature maps. We will implement this network using the *TensorFlow Keras API*.

Loading and preprocessing the data

The three steps for loading the **MNIST** dataset are as follows:

```
import tensorflow_datasets as tfds
import pandas as pd

import matplotlib.pyplot as plt
%matplotlib inline

## MNIST dataset

mnist_bldr = tfds.builder('mnist')
mnist_bldr.download_and_prepare()
datasets = mnist_bldr.as_dataset(shuffle_files=False)
print(datasets.keys())
mnist_train_orig, mnist_test_orig = datasets['train'],
datasets['test']

dict_keys(['test', 'train'])
```

The MNIST dataset comes with a pre-specified training and test dataset partitioning scheme, but we also want to create a validation split from the train partition. Notice that in the third step, we used an optional argument, `shuffle_files=False`, in the `.as_dataset()`

method. This prevented initial shuffling, which is necessary for us since we want to split the training dataset into two parts: a smaller training dataset and a validation dataset. (Note: if the initial shuffling was not turned off, it would incur reshuffling of the dataset every time we fetched a mini-batch of data.)

We can split the train/validation datasets as follows:

```
BUFFER_SIZE = 10000
BATCH_SIZE = 64
NUM_EPOCHS = 20

mnist_train = mnist_train_orig.map(
    lambda item: (tf.cast(item['image'], tf.float32)/255.0,
                  tf.cast(item['label'], tf.int32)))

mnist_test = mnist_test_orig.map(
    lambda item: (tf.cast(item['image'], tf.float32)/255.0,
                  tf.cast(item['label'], tf.int32)))

tf.random.set_seed(1)

mnist_train = mnist_train.shuffle(buffer_size=BUFFER_SIZE,
                                 reshuffle_each_iteration=False)

mnist_valid = mnist_train.take(10000).batch(BATCH_SIZE)
mnist_train = mnist_train.skip(10000).batch(BATCH_SIZE)
```

Implementing a CNN using the TensorFlow Keras API

For implementing a CNN in TensorFlow, we use the Keras `Sequential` class to stack different layers, such as convolution, pooling, and dropout, as well as the fully connected (*dense*) layers. The Keras layers API provides classes for each one:

`tf.keras.layers.Conv2D` for a two-dimensional convolution layer;
`tf.keras.layers.MaxPool2D` and `tf.keras.layers.AvgPool2D` for subsampling (maxpooling and average-pooling); and `tf.keras.layers.Dropout` for regularization using dropout. We will go over each of these classes in more detail.

Configuring CNN layers in Keras

Constructing a layer with the `Conv2D` class requires us to specify the number of output filters (which is equivalent to the number of output feature maps) and kernel sizes. In addition, there are optional parameters that we can use to configure a convolutional layer. The most commonly used ones are the strides (with a default value of 1 in both *x*, *y* dimensions) and padding, which could be same or valid.

- **Conv2D:** `tf.keras.layers.Conv2D`
 - `filters`
 - `kernel_size`
 - `strides`
 - `padding`
- **MaxPool2D:** `tf.keras.layers.MaxPool2D`
 - `pool_size`
 - `strides`

- padding
- **Dropout** `tf.keras.layers.Dropout2D`
 - rate

Additional configuration parameters are listed in the [official documentation](#).

Constructing a CNN in Keras

Now we can construct the CNN model that was shown in the previous figure. In the following code, we will use the `Sequential` class and add the convolution and pooling layers:

```
model = tf.keras.Sequential()

model.add(tf.keras.layers.Conv2D(
    filters=32, kernel_size=(5, 5),
    strides=(1, 1), padding='same',
    data_format='channels_last',
    name='conv_1', activation='relu'))

model.add(tf.keras.layers.MaxPool2D(
    pool_size=(2, 2), name='pool_1'))

model.add(tf.keras.layers.Conv2D(
    filters=64, kernel_size=(5, 5),
    strides=(1, 1), padding='same',
    name='conv_2', activation='relu'))

model.add(tf.keras.layers.MaxPool2D(pool_size=(2, 2),
    name='pool_2'))
```

So far, we have added two convolution layers to the model. For each convolutional layer, we used a kernel of size 5×5 and 'same' padding. As discussed earlier, using `padding='same'` preserves the spatial dimensions (vertical and horizontal dimensions) of the feature maps such that the inputs and outputs have the same height and width (and the number of channels may only differ in terms of the number of filters used). The max-pooling layers with pooling size 2×2 and strides of 2 will reduce the spatial dimensions by half. (Note that if the `strides` parameter is not specified in `MaxPool2D`, by default, it is set equal to the pooling size.) While we can calculate the size of the feature maps at this stage manually, the Keras API provides a convenient method to compute this for us:

```
model.compute_output_shape(input_shape=(16, 28, 28, 1))

TensorShape([16, 7, 7, 64])
```

By providing the input shape as a tuple specified in this example, the method `compute_output_shape` calculated the output to have a shape `(16, 7, 7, 64)`, indicating feature maps with 64 channels and a spatial size of 7×7 . The first dimension corresponds to the batch dimension, for which we used 16 arbitrarily. We could have used `None` instead, that is, `input_shape=(None, 28, 28, 1)`. The next layer that we want to add is a dense (or fully connected) layer for implementing a classifier on top of our convolutional and pooling layers. The input to this layer must have rank 2, that is, shape `[batchsize × input_units]`. Thus, we need to flatten the output of the previous layers to meet this requirement for the dense layer:

```

model.add(tf.keras.layers.Flatten())

model.compute_output_shape(input_shape=(16, 28, 28, 1))

TensorShape([16, 3136])

```

As the result of `compute_output_shape` indicates, the input dimensions for the dense layer are correctly set up. Next, we will add two dense layers with a dropout layer in between:

```

model.add(tf.keras.layers.Dense(
    units=1024, name='fc_1',
    activation='relu'))

model.add(tf.keras.layers.Dropout(
    rate=0.5))

model.add(tf.keras.layers.Dense(
    units=10, name='fc_2',
    activation='softmax'))

```

The last fully connected layer, named `fc_2`, has 10 output units for the 10 class labels in the **MNIST** dataset. Also, we use the softmax activation to obtain the classmembership probabilities of each input example, assuming that the classes are mutually exclusive, so the probabilities for each example sum to 1. (This means that a training example can belong to only one class.) Based on what we discussed in the section Loss functions for classification, which loss should we use here? Remember that for a multiclass classification with integer (sparse) labels (as opposed to one-hot encoded labels), we use `SparseCategoricalCrossentropy`. The following code will call the `build()` method for late variable creation and compile the model:

```

tf.random.set_seed(1)
model.build(input_shape=(None, 28, 28, 1))

model.compute_output_shape(input_shape=(16, 28, 28, 1))

TensorShape([16, 10])

model.summary()

Model: "sequential"

```

Layer (type)	Output Shape	Param #
=====		
conv_1 (Conv2D)	multiple	832
pool_1 (MaxPooling2D)	multiple	0
conv_2 (Conv2D)	multiple	51264
pool_2 (MaxPooling2D)	multiple	0
flatten (Flatten)	multiple	0

fc_1 (Dense)	multiple	3212288
dropout (Dropout)	multiple	0
fc_2 (Dense)	multiple	10250
=====		
Total params: 3,274,634		
Trainable params: 3,274,634		
Non-trainable params: 0		

As you already know, we can train the model by calling the `fit()` method. Note that using the designated methods for training and evaluation (like `evaluate()` and `predict()`) will automatically set the mode for the dropout layer and rescale the hidden units appropriately so that we do not have to worry about that at all. Next, we will train this CNN model and use the validation dataset that we created for monitoring the learning progress:

```
model.compile(optimizer=tf.keras.optimizers.Adam(),
              loss=tf.keras.losses.SparseCategoricalCrossentropy(),
              metrics=['accuracy']) # same as
`tf.keras.metrics.SparseCategoricalAccuracy(name='accuracy')`
```

```
history = model.fit(mnist_train, epochs=NUM_EPOCHS,
                   validation_data=mnist_valid,
                   shuffle=True)
```

```
Epoch 1/20
782/782 [=====] - 50s 64ms/step - loss:
0.1453 - accuracy: 0.9548 - val_loss: 0.0000e+00 - val_accuracy:
0.0000e+00
Epoch 2/20
782/782 [=====] - 53s 67ms/step - loss:
0.0489 - accuracy: 0.9857 - val_loss: 0.0375 - val_accuracy: 0.9879
Epoch 3/20
782/782 [=====] - 58s 75ms/step - loss:
0.0318 - accuracy: 0.9904 - val_loss: 0.0436 - val_accuracy: 0.9863
Epoch 4/20
782/782 [=====] - 56s 71ms/step - loss:
0.0236 - accuracy: 0.9925 - val_loss: 0.0627 - val_accuracy: 0.9822
Epoch 5/20
782/782 [=====] - 56s 72ms/step - loss:
0.0192 - accuracy: 0.9942 - val_loss: 0.0340 - val_accuracy: 0.9912
Epoch 6/20
782/782 [=====] - 57s 73ms/step - loss:
0.0163 - accuracy: 0.9953 - val_loss: 0.0516 - val_accuracy: 0.9866
Epoch 7/20
782/782 [=====] - 57s 73ms/step - loss:
0.0161 - accuracy: 0.9954 - val_loss: 0.0375 - val_accuracy: 0.9908
Epoch 8/20
782/782 [=====] - 58s 74ms/step - loss:
0.0109 - accuracy: 0.9962 - val_loss: 0.0392 - val_accuracy: 0.9901
Epoch 9/20
```

```

782/782 [=====] - 56s 72ms/step - loss:
0.0112 - accuracy: 0.9966 - val_loss: 0.0425 - val_accuracy: 0.9893
Epoch 10/20
782/782 [=====] - 56s 72ms/step - loss:
0.0090 - accuracy: 0.9971 - val_loss: 0.0418 - val_accuracy: 0.9908
Epoch 11/20
782/782 [=====] - 56s 72ms/step - loss:
0.0103 - accuracy: 0.9968 - val_loss: 0.0458 - val_accuracy: 0.9902
Epoch 12/20
782/782 [=====] - 58s 74ms/step - loss:
0.0088 - accuracy: 0.9971 - val_loss: 0.0506 - val_accuracy: 0.9899
Epoch 13/20
782/782 [=====] - 57s 73ms/step - loss:
0.0065 - accuracy: 0.9978 - val_loss: 0.0385 - val_accuracy: 0.9911
Epoch 14/20
782/782 [=====] - 56s 71ms/step - loss:
0.0063 - accuracy: 0.9980 - val_loss: 0.0432 - val_accuracy: 0.9907
Epoch 15/20
782/782 [=====] - 60s 76ms/step - loss:
0.0071 - accuracy: 0.9980 - val_loss: 0.0606 - val_accuracy: 0.9885
Epoch 16/20
782/782 [=====] - 59s 75ms/step - loss:
0.0062 - accuracy: 0.9981 - val_loss: 0.0675 - val_accuracy: 0.9875
Epoch 17/20
782/782 [=====] - 61s 78ms/step - loss:
0.0073 - accuracy: 0.9980 - val_loss: 0.0434 - val_accuracy: 0.9917
Epoch 18/20
782/782 [=====] - 59s 75ms/step - loss:
0.0051 - accuracy: 0.9983 - val_loss: 0.0585 - val_accuracy: 0.9894
Epoch 19/20
782/782 [=====] - 59s 75ms/step - loss:
0.0068 - accuracy: 0.9982 - val_loss: 0.0550 - val_accuracy: 0.9917
Epoch 20/20
782/782 [=====] - 58s 74ms/step - loss:
0.0076 - accuracy: 0.9981 - val_loss: 0.0573 - val_accuracy: 0.9906

```

Once the 20 epochs of training are finished, we can visualize the learning curves:

```

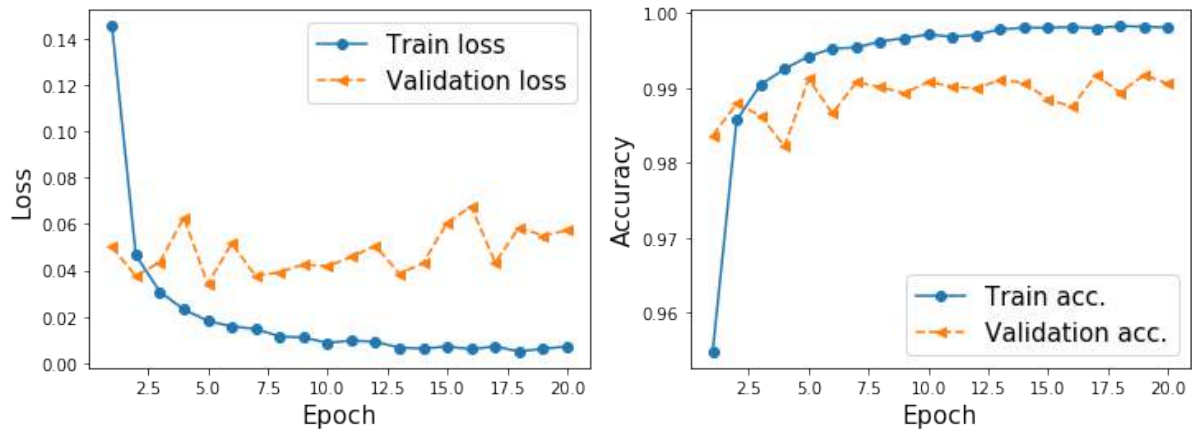
hist = history.history
x_arr = np.arange(len(hist['loss'])) + 1

fig = plt.figure(figsize=(12, 4))
ax = fig.add_subplot(1, 2, 1)
ax.plot(x_arr, hist['loss'], '-o', label='Train loss')
ax.plot(x_arr, hist['val_loss'], '--<', label='Validation loss')
ax.set_xlabel('Epoch', size=15)
ax.set_ylabel('Loss', size=15)
ax.legend(fontsize=15)
ax = fig.add_subplot(1, 2, 2)
ax.plot(x_arr, hist['accuracy'], '-o', label='Train acc.')
ax.plot(x_arr, hist['val_accuracy'], '--<', label='Validation acc.')
ax.legend(fontsize=15)
ax.set_xlabel('Epoch', size=15)

```

```
ax.set_ylabel('Accuracy', size=15)

#plt.savefig('figures/15_12.png', dpi=300)
plt.show()
```



As you already know, evaluating the trained model on the test dataset can be done by calling the `.evaluate()` method:

```
test_results = model.evaluate(mnist_test.batch(20))
print('\nTest Acc. {:.2f}%'.format(test_results[1]*100))
```

500/500 [=====] - 5s 10ms/step - loss: 0.0443 - accuracy: 0.9929

Test Acc. 99.29%

The CNN model achieves an accuracy near 99%. Finally, we can get the prediction results in the form of class-membership probabilities and convert them to predicted labels by using the `tf.argmax` function to find the element with the maximum probability. We will do this for a batch of 12 examples and visualize the input and predicted labels:

```
batch_test = next(iter(mnist_test.batch(12)))

preds = model(batch_test[0])

tf.print(preds.shape)
preds = tf.argmax(preds, axis=1)
print(preds)

fig = plt.figure(figsize=(12, 4))
for i in range(12):
    ax = fig.add_subplot(2, 6, i+1)
    ax.set_xticks([]); ax.set_yticks([])
    img = batch_test[0][i, :, :, 0]
    ax.imshow(img, cmap='gray_r')
    ax.text(0.9, 0.1, '{}'.format(preds[i]),
           size=15, color='blue',
           horizontalalignment='center',
           verticalalignment='center',
```

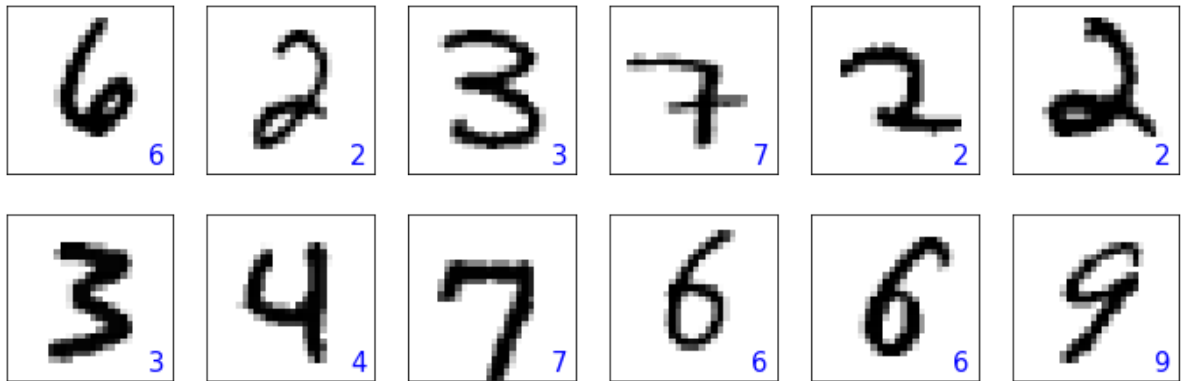
```

transform=ax.transAxes)

#plt.savefig('figures/15_13.png', dpi=300)
plt.show()

TensorShape([12, 10])
tf.Tensor([6 2 3 7 2 2 3 4 7 6 6 9], shape=(12,), dtype=int64)

```



In this set of plotted examples, all the predicted labels are correct.

```

import os

if not os.path.exists('models'):
    os.mkdir('models')

model.save('models/mnist-cnn.h5')

```

Neural Network in AI. Part 3

```
from google.colab import auth
auth.authenticate_user()

!pip install google-cloud-storage

# Authenticate to GCS.
from google.colab import auth
auth.authenticate_user()

# Create the service client.
from googleapiclient.discovery import build
gcs_service = build('storage', 'v1')

from IPython.display import Image
%matplotlib inline

from google.colab import drive
drive.mount('/content/drive')

%cd /content/drive/My Drive/Colab Notebooks/AI Course/

import tensorflow as tf
import tensorflow_datasets as tfds
import numpy as np

from IPython.display import Image

import matplotlib.pyplot as plt
%matplotlib inline
```

1. Gender classification from face images using CNN

In this section, we are going to implement a CNN for gender classification from face images using the **CelebA** dataset. The CelebA dataset contains 202,599 face images of celebrities. In addition, 40 binary facial attributes are available for each image, including gender (male or female) and age (young or old).

Based on what you have learned so far, the goal of this section is to build and train a CNN model for predicting the gender attribute from these face images. Here, for simplicity, we will only be using a small portion of the training data (16,000 training examples) to speed up the training process. However, in order to improve the generalization performance and reduce overfitting on such a small dataset, we will use a technique called **data augmentation**.

Loading the CelebA dataset

First, let's load the data similarly to how we did in the previous section for the MNIST dataset. CelebA data comes in three partitions: a training dataset, a validation dataset, and a test dataset.

Moreover, we can implement a simple function to count the number of examples in each partition:

```
gcs_base_dir = "gs://celeb_a_dataset/"
celeba_bldr = tfds.builder("celeb_a", data_dir=gcs_base_dir,
version='2.1.0')
celeba_bldr.download_and_prepare()

celeba = celeba_bldr.as_dataset(shuffle_files=False)
print(celeba.keys())

celeba_train = celeba['train']
celeba_valid = celeba['validation']
celeba_test = celeba['test']

def count_items(ds):
    n = 0
    for _ in ds:
        n += 1
    return n

#print('Train set: {}'.format(count_items(celeba_train)))
#print('Validation: {}'.format(count_items(celeba_valid)))
#print('Test set: {}'.format(count_items(celeba_test)))
```

So, instead of using all the available training and validation data, we will take a subset of 16,000 training examples and 1,000 examples for validation, as follows:

```
celeba_train = celeba_train.take(16000)
celeba_valid = celeba_valid.take(1000)

#print('Train set: {}'.format(count_items(celeba_train)))
#print('Validation: {}'.format(count_items(celeba_valid)))
```

It is important to note that if the `shuffle_files` argument in `celeba_bldr.as_dataset()` was not set to `False`, we would still see 16,000 examples in the training dataset and 1,000 examples for the validation dataset. However, at each iteration, it would reshuffle the training data and take a new set of 16,000 examples. This would defeat the purpose, as our goal here is to intentionally train our model with a small dataset. Next, we will discuss data augmentation as a technique for boosting the performance of deep NNs.

Image transformation and data augmentation

Data augmentation summarizes a broad set of techniques for dealing with cases where the training data is limited. For instance, certain data augmentation techniques allow us to modify or even artificially synthesize more data and thereby boost the performance of a machine or deep learning model by reducing overfitting. While data augmentation is not only

for image data, there are a set of transformations uniquely applicable to image data, such as cropping parts of an image, flipping, changing the contrast, brightness, and saturation. Let's see some of these transformations that are available via the `tf.image` module. In the following code block, we will first get five examples from the `celeba_train` dataset and apply five different types of transformation:

1. cropping an image to a bounding box,
2. flipping an image horizontally,
3. adjusting the contrast,
4. adjusting the brightness, and
5. center-cropping an image and resizing the resulting image back to its original size, (218, 178).

In the following code, we will visualize the results of these transformations, showing each one in a separate column for comparison:

```
## take 5 examples:
examples = []
for example in celeba_train.take(5):
    examples.append(example['image'])

fig = plt.figure(figsize=(16, 8.5))

## Column 1: cropping to a bounding-box
ax = fig.add_subplot(2, 5, 1)
ax.imshow(examples[0])
ax = fig.add_subplot(2, 5, 6)
ax.set_title('Crop to a \nbounding-box', size=15)
img_cropped = tf.image.crop_to_bounding_box(
    examples[0], 50, 20, 128, 128)
ax.imshow(img_cropped)

## Column 2: flipping (horizontally)
ax = fig.add_subplot(2, 5, 2)
ax.imshow(examples[1])
ax = fig.add_subplot(2, 5, 7)
ax.set_title('Flip (horizontal)', size=15)
img_flipped = tf.image.flip_left_right(examples[1])
ax.imshow(img_flipped)

## Column 3: adjust contrast
ax = fig.add_subplot(2, 5, 3)
ax.imshow(examples[2])
ax = fig.add_subplot(2, 5, 8)
ax.set_title('Adjust contrast', size=15)
img_adj_contrast = tf.image.adjust_contrast(
    examples[2], contrast_factor=2)
ax.imshow(img_adj_contrast)

## Column 4: adjust brightness
ax = fig.add_subplot(2, 5, 4)
ax.imshow(examples[3])
ax = fig.add_subplot(2, 5, 9)
ax.set_title('Adjust brightness', size=15)
```

```

img_adj_brightness = tf.image.adjust_brightness(
    examples[3], delta=0.3)
ax.imshow(img_adj_brightness)

## Column 5: cropping from image center
ax = fig.add_subplot(2, 5, 5)
ax.imshow(examples[4])
ax = fig.add_subplot(2, 5, 10)
ax.set_title('Central crop and resize', size=15)
img_center_crop = tf.image.central_crop(
    examples[4], 0.7)
img_resized = tf.image.resize(
    img_center_crop, size=(218, 178))
ax.imshow(img_resized.numpy().astype('uint8'))

# plt.savefig('figures/15_14.png', dpi=300)
plt.show()

```

In the previous figure, the original images are shown in the first row and their transformed version in the second row. Note that for the first transformation (*leftmost column*), the bounding box is specified by four numbers: the coordinate of the upper-left corner of the bounding box (here $x=20$, $y=50$), and the width and height of the box ($width=128$, $height=128$). Also note that the origin (the coordinates at the location denoted as $(0, 0)$) for images loaded by TensorFlow (as well as other packages such as imageio) is the upper-left corner of the image.

The transformations in the previous code block are deterministic. However, all such transformations can also be randomized, which is recommended for data augmentation during model training. For example, a random bounding box (where the coordinates of the upper-left corner are selected randomly) can be cropped from an image, an image can be randomly flipped along either the horizontal or vertical axes with a probability of 0.5, or the contrast of an image can be changed randomly, where the `contrast_factor` is selected at random, but with uniform distribution, from a range of values. In addition, we can create a pipeline of these transformations.

For example, we can first randomly crop an image, then flip it randomly, and finally, resize it to the desired size. The code is as follows (since we have random elements, we set the random seed for reproducibility):

```

tf.random.set_seed(1)

fig = plt.figure(figsize=(14, 12))

for i, example in enumerate(celeba_train.take(3)):
    image = example['image']

    ax = fig.add_subplot(3, 4, i*4+1)
    ax.imshow(image)
    if i == 0:
        ax.set_title('Orig.', size=15)

    ax = fig.add_subplot(3, 4, i*4+2)
    img_crop = tf.image.random_crop(image, size=(178, 178, 3))

```

```

ax.imshow(img_crop)
if i == 0:
    ax.set_title('Step 1: Random crop', size=15)

ax = fig.add_subplot(3, 4, i*4+3)
img_flip = tf.image.random_flip_left_right(img_crop)
ax.imshow(tf.cast(img_flip, tf.uint8))
if i == 0:
    ax.set_title('Step 2: Random flip', size=15)

ax = fig.add_subplot(3, 4, i*4+4)
img_resize = tf.image.resize(img_flip, size=(128, 128))
ax.imshow(tf.cast(img_resize, tf.uint8))
if i == 0:
    ax.set_title('Step 3: Resize', size=15)

# plt.savefig('figures/15_15.png', dpi=300)
plt.show()

```

Note that each time we iterate through these three examples, we get slightly different images due to random transformations. For convenience, we can define a wrapper function to use this pipeline for data augmentation during model training. In the following code, we will define the function `preprocess()`, which will receive a dictionary containing the keys 'image' and 'attributes'. The function will return a tuple containing the transformed image and the label extracted from the dictionary of attributes.

We will only apply data augmentation to the training examples, however, and not to the validation or test images. The code is as follows:

```

def preprocess(example, size=(64, 64), mode='train'):
    image = example['image']
    label = example['attributes']['Male']
    if mode == 'train':
        image_cropped = tf.image.random_crop(
            image, size=(178, 178, 3))
        image_resized = tf.image.resize(
            image_cropped, size=size)
        image_flip = tf.image.random_flip_left_right(
            image_resized)
        return (image_flip/255.0, tf.cast(label, tf.int32))

    else:
        image_cropped = tf.image.crop_to_bounding_box(
            image, offset_height=20, offset_width=0,
            target_height=178, target_width=178)
        image_resized = tf.image.resize(
            image_cropped, size=size)
        return (image_resized/255.0, tf.cast(label, tf.int32))

## testing:
#item = next(iter(celeba_train))
#preprocess(item, mode='train')

```

Now, to see data augmentation in action, let's create a small subset of the training dataset, apply this function to it, and iterate over the dataset five times:

```
tf.random.set_seed(1)

ds = celeba_train.shuffle(1000, reshuffle_each_iteration=False)
ds = ds.take(2).repeat(5)

ds = ds.map(lambda x: preprocess(x, size=(178, 178), mode='train'))

fig = plt.figure(figsize=(15, 6))
for j, example in enumerate(ds):
    ax = fig.add_subplot(2, 5, j//2+(j%2)*5+1)
    ax.set_xticks([])
    ax.set_yticks([])
    ax.imshow(example[0])

#plt.savefig('figures/15_16.png', dpi=300)
plt.show()
```

This figure shows the five resulting transformations for data augmentation on two example images.

Next, we will apply this preprocessing function to our training and validation datasets. We will use an image size of (64, 64). Furthermore, we will specify `mode='train'` when working with training data and use `mode='eval'` for the validation data so that the random elements of the data augmentation pipeline will be applied only to the training data:

```
BATCH_SIZE = 32
BUFFER_SIZE = 1000
IMAGE_SIZE = (64, 64)
steps_per_epoch = np.ceil(16000/BATCH_SIZE)
print(steps_per_epoch)

ds_train = celeba_train.map(
    lambda x: preprocess(x, size=IMAGE_SIZE, mode='train'))
ds_train = ds_train.shuffle(buffer_size=BUFFER_SIZE).repeat()
ds_train = ds_train.batch(BATCH_SIZE)

ds_valid = celeba_valid.map(
    lambda x: preprocess(x, size=IMAGE_SIZE, mode='eval'))
ds_valid = ds_valid.batch(BATCH_SIZE)
```

Training a CNN gender classifier

By now, building a model with TensorFlow's Keras API and training it should be straightforward. The design of our CNN is as follows: the CNN model receives input images of size $64 \times 64 \times 3$ (the images have three color channels, using `'channels_last'`).

The input data goes through four convolutional layers to make 32, 64, 128, and 256 feature maps using filters with kernel a size of 3×3 . The first three convolution layers are followed by max-pooling, $P_{2 \times 2}$. Two dropout layers are also included for regularization:

```

model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(
        32, (3, 3), padding='same', activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Dropout(rate=0.5),

    tf.keras.layers.Conv2D(
        64, (3, 3), padding='same', activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Dropout(rate=0.5),

    tf.keras.layers.Conv2D(
        128, (3, 3), padding='same', activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),

    tf.keras.layers.Conv2D(
        256, (3, 3), padding='same', activation='relu'),
])

```

Let's see the shape of the output feature maps after applying these layers:

```

model.compute_output_shape(input_shape=(None, 64, 64, 3))

```

There are 256 feature maps (or channels) of size 8×8 . Now, we can add a fully connected layer to get to the output layer with a single unit. If we reshape (flatten) the feature maps, the number of input units to this fully connected layer will be $8 \times 8 \times 256 = 16,384$.

Alternatively, let's consider a new layer, called **global average pooling**, which computes the average of each feature map separately, thereby reducing the hidden units to 256. We can then add a fully connected layer. Although we have not discussed global average-pooling explicitly, it is conceptually very similar to other pooling layers. Global average-pooling can be viewed, in fact, as a special case of average-pooling when the pooling size is equal to the size of the input feature maps.

To understand this, consider the following figure showing an example of input feature maps of shape $[batchsize \times 64 \times 64 \times 8]$. The channels are numbered $k=0, 1, \dots, 7$. The global average-pooling operation calculates the average of each channel so that the output will have shape $[batchsize \times 8]$. (Note: `GlobalAveragePooling2D` in the Keras API will automatically squeeze the output.

Without squeezing the output, the shape would be $[batchsize \times 1 \times 1 \times 8]$, as the global average-pooling would reduce the spatial dimension of 64×64 to a 1×1 :

```

Image(filename='images/15_13.png', width=800)

```

Therefore, given that, in our case, the shape of the feature maps prior to this layer is $[batchsize \times 8 \times 8 \times 256]$, we expect to get 256 units as output, that is, the shape of the output will be $[batchsize \times 256]$. Let's add this layer and recompute the output shape to verify that this is true:

```

model.add(tf.keras.layers.GlobalAveragePooling2D())
model.compute_output_shape(input_shape=(None, 64, 64, 3))

```

Finally, we can add a fully connected (dense) layer to get a single output unit. In this case, we can specify the activation function to be 'sigmoid' or just use `activation=None`, so that the model will output the logits (instead of classmembership probabilities), which is preferred for model training in TensorFlow and Keras due to numerical stability, as discussed earlier:

```
model.add(tf.keras.layers.Dense(1, activation=None))
```

The next step is to compile the model and, at this time, we have to decide what loss function to use. We have a binary classification with a single output unit, so that means we should use `BinaryCrossentropy`. In addition, since our last layer does not apply sigmoid activation (we used `activation=None`), the outputs of our model are logits, not probabilities. Therefore, we will also specify `from_logits=True` in `BinaryCrossentropy` so that the loss function applies the sigmoid function internally, which is, due to the underlying code, more efficient than doing it manually. The code for compiling and training the model is as follows:

```
tf.random.set_seed(1)

model.build(input_shape=(None, 64, 64, 3))

model.summary()

model.compile(optimizer=tf.keras.optimizers.Adam(),
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = model.fit(ds_train, validation_data=ds_valid,
                   epochs=20, steps_per_epoch=steps_per_epoch)
```

Let's now visualize the learning curve and compare the training and validation loss and accuracies after each epoch:

```
hist = history.history
x_arr = np.arange(len(hist['loss'])) + 1

fig = plt.figure(figsize=(12, 4))
ax = fig.add_subplot(1, 2, 1)
ax.plot(x_arr, hist['loss'], '-o', label='Train loss')
ax.plot(x_arr, hist['val_loss'], '--<', label='Validation loss')
ax.legend(fontsize=15)
ax.set_xlabel('Epoch', size=15)
ax.set_ylabel('Loss', size=15)

ax = fig.add_subplot(1, 2, 2)
ax.plot(x_arr, hist['accuracy'], '-o', label='Train acc.')
ax.plot(x_arr, hist['val_accuracy'], '--<', label='Validation acc.')
ax.legend(fontsize=15)
ax.set_xlabel('Epoch', size=15)
ax.set_ylabel('Accuracy', size=15)
```

```
#plt.savefig('figures/15_18.png', dpi=300)
plt.show()
```

The figure shows the training and validation losses and accuracies.

As you can see from the learning curves, the losses for the training and validation have not converged to a plateau region. Based on this result, we could have continued training for a few more epochs. Using the `fit()` method, we can continue training for an additional 10 epochs.

Once we are happy with the learning curves, we can evaluate the model on the holdout test dataset.

```
ds_test = celeba_test.map(
    lambda x: preprocess(x, size=IMAGE_SIZE, mode='eval')).batch(32)
results = model.evaluate(ds_test, verbose=0)
print('Test Acc: {:.2f}%'.format(results[1]*100))

history = model.fit(ds_train, validation_data=ds_valid,
                    epochs=30, initial_epoch=20,
                    steps_per_epoch=steps_per_epoch)

hist2 = history.history
x_arr = np.arange(len(hist['loss'] + hist2['loss']))

fig = plt.figure(figsize=(12, 4))
ax = fig.add_subplot(1, 2, 1)
ax.plot(x_arr, hist['loss']+hist2['loss'],
        '-o', label='Train Loss')
ax.plot(x_arr, hist['val_loss']+hist2['val_loss'],
        '--<', label='Validation Loss')
ax.legend(fontsize=15)

ax = fig.add_subplot(1, 2, 2)
ax.plot(x_arr, hist['accuracy']+hist2['accuracy'],
        '-o', label='Train Acc.')
ax.plot(x_arr, hist['val_accuracy']+hist2['val_accuracy'],
        '--<', label='Validation Acc.')
ax.legend(fontsize=15)
plt.show()

ds_test = celeba_test.map(
    lambda x: preprocess(x, size=IMAGE_SIZE, mode='eval')).batch(32)
results = model.evaluate(ds_test, verbose=0)
print('Test Acc: {:.2f}%'.format(results[1]*100))
```

Finally, we already know how to get the prediction results on some test examples using `model.predict()`. However, remember that the model outputs the logits, not probabilities. If we are interested in the class-membership probabilities for this binary problem with a single output unit, we can use the `tf.sigmoid` function to compute the probability of class 1. (In the case of a multiclass problem, we would use `tf.math.softmax`.) In the following code, we will take a small subset of 10 examples from

our pre-processed test dataset (`ds_test`) and run `model.predict()` to get the logits. Then, we will compute the probabilities of each example being from class 1 (which corresponds to male based on the labels provided in CelebA) and visualize the examples along with their ground truth label and the predicted probabilities.

Notice that we first apply `unbatch()` to the `ds_test` dataset before taking 10 examples; otherwise, the `take()` method would return 10 batches of size 32, instead of 10 individual examples:

```
ds = ds_test.unbatch().take(10)

pred_logits = model.predict(ds.batch(10))
probas = tf.sigmoid(pred_logits)
probas = probas.numpy().flatten()*100

fig = plt.figure(figsize=(15, 7))
for j, example in enumerate(ds):
    ax = fig.add_subplot(2, 5, j+1)
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(example[0])
    if example[1].numpy() == 1:
        label='Male'
    else:
        label = 'Female'
    ax.text(
        0.5, -0.15,
        'GT: {:s}\nPr(Male)={:.0f}%'.format(label, probas[j]),
        size=16,
        horizontalalignment='center',
        verticalalignment='center',
        transform=ax.transAxes)

#plt.savefig('figures/figures-15_19.png', dpi=300)
plt.show()
```

In the figure, you can see 10 example images along with their ground truth labels and the probabilities that they belong to class 1, male.

The probabilities of class 1 (that is, male according to CelebA) are provided below each image. As you can see, our trained model made only one error on this set of 10 test examples. As an optional exercise, you are encouraged to try using the entire training dataset instead of the small subset we created. Furthermore, you can change or modify the CNN architecture. For example, you can change the dropout probabilities and the number of filters in the different convolutional layers. Also, you could replace the global average-pooling with a dense layer. If you are using the entire training dataset with the CNN architecture, you should be able to achieve about 97-99% accuracy.

3. Modeling Sequential Data Using Recurrent Neural Networks

Introducing sequential data

Modeling sequential data—order matters

What makes sequences unique, compared to other types of data, is that elements in a sequence appear in a certain order and are not independent of each other. Typical machine learning algorithms for supervised learning assume that the input is **independent and identically distributed (IID) data**, which means that the training examples are mutually independent and have the same underlying distribution. In this regard, based on the mutual independence assumption, the order in which the training examples are given to the model is irrelevant. For example, if we have a sample consisting of n training examples, $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(n)}$, the order in which we use the data for training our machine learning algorithm does not matter. An example of this scenario would be the Iris dataset that we previously worked with. In the Iris dataset, each flower has been measured independently, and the measurements of one flower do not influence the measurements of another flower.

However, this assumption is not valid when we deal with sequences—by definition, order matters. Predicting the market value of a particular stock would be an example of this scenario. For instance, assume we have a sample of n training examples, where each training example represents the market value of a certain stock on a particular day. If our task is to predict the stock market value for the next three days, it would make sense to consider the previous stock prices in a date-sorted order to derive trends rather than utilize these training examples in a randomized order.

Representing sequences

We've established that order among data points is important in sequential data, so we next need to find a way to leverage this ordering information in a machine learning model. Throughout this chapter, we will represent sequences as $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(T)}$. The superscript indices indicate the order of the instances, and the length of the sequence is T . For a sensible example of sequences, consider time-series data, where each example point, $\mathbf{x}^{(t)}$, belongs to a particular time, t . The following figure shows an example of time-series data where both the input features ($\mathbf{x}$) and the target labels ($\mathbf{y}$) naturally follow the order according to their time axis; therefore, both the $\mathbf{x}$'s and $\mathbf{y}$'s are sequences:

```
Image(filename='images/16_01.png', width=700)
```

As we have already mentioned, the standard neural network (NN) models that we have covered so far, such as multilayer perceptron (MLP) and CNNs for image data, assume that the training examples are independent of each other and thus do not incorporate ordering information. We can say that such models do not have a memory of previously seen training examples. For instance, the samples are passed through the feedforward and backpropagation steps, and the weights are updated independently of the order in which the training examples are processed.

RNNs, by contrast, are designed for modeling sequences and are capable of remembering past information and processing new events accordingly, which is a clear advantage when working with sequence data.

The different categories of sequence modeling

Sequence modeling has many fascinating applications, such as language translation (for example, translating text from English to German), image captioning, and text generation. However, in order to choose an appropriate architecture and approach, we have to understand and be able to distinguish between these different sequence modeling tasks. The following figure, based on the explanations in the excellent article [The Unreasonable Effectiveness of Recurrent Neural Networks](#), by Andrej Karpathy, summarizes the most

common sequence modeling tasks, which depend on the relationship categories of input and output data:

```
Image(filename='images/16_02.png', width=700)
```

Let's discuss the different relationship categories between input and output data, which were depicted in the previous figure, in more detail. If neither the input nor output data represents sequences, then we are dealing with standard data, and we could simply use a multilayer perceptron (or another classification model previously covered in this book) to model such data. However, if either the input or output is a sequence, the modeling task likely falls into one of these categories:

- *Many-to-one*: The input data is a sequence, but the output is a fixed-size vector or scalar, not a sequence. For example, in sentiment analysis, the input is text-based (for example, a movie review) and the output is a class label (for example, a label denoting whether a reviewer liked the movie).
- *One-to-many*: The input data is in standard format and not a sequence, but the output is a sequence. An example of this category is image captioning—the input is an image and the output is an English phrase summarizing the content of that image.
- *Many-to-many*: Both the input and output arrays are sequences. This category can be further divided based on whether the input and output are synchronized. An example of a synchronized many-to-many modeling task is video classification, where each frame in a video is labeled. An example of a delayed many-to-many modeling task would be translating one language into another. For instance, an entire English sentence must be read and processed by a machine before its translation into German is produced.

RNNs for modeling sequences

Understanding the RNN looping mechanism

Let's start with the architecture of an RNN. The following figure shows a standard feedforward NN and an RNN side by side for comparison:

```
Image(filename='images/16_03.png', width=700)
```

Both of these networks have only one hidden layer. In this representation, the units are not displayed, but we assume that the input layer (x), hidden layer (h), and output layer (o) are vectors that contain many units.

In a standard feedforward network, information flows from the input to the hidden layer, and then from the hidden layer to the output layer. On the other hand, in an RNN, the hidden layer receives its input from both the input layer of the current time step and the hidden layer from the previous time step. The flow of information in adjacent time steps in the hidden layer allows the network to have a memory of past events. This flow of information is usually displayed as a loop, also known as a **recurrent edge** in graph notation, which is how this general RNN architecture got its name. Similar to multilayer perceptrons, RNNs can consist of multiple hidden layers. Note that it's a common convention to refer to RNNs with one hidden layer as a *singlelayer RNN*, which is not to be confused with single-layer NNs without a hidden layer, such as Adaline or logistic regression. The following figure illustrates an RNN with one hidden layer (top) and an RNN with two hidden layers (bottom):

```
Image(filename='images/16_04.png', width=700)
```

As we know, each hidden unit in a standard NN receives only one input—the net preactivation associated with the input layer. In contrast, each hidden unit in an RNN receives two distinct sets of input—the preactivation from the input layer and the activation of the same hidden layer from the previous time step, $t - 1$. At the first time step, $t = 0$, the hidden units are initialized to zeros or small random values. Then, at a time step where $t > 0$, the hidden units receive their input from the data point at the current time, $\mathbf{x}^{(t)}$, and the previous values of hidden units at $t - 1$, indicated as $\mathbf{h}^{(t-1)}$.

Similarly, in the case of a multilayer RNN, we can summarize the information flow as follows:

- *layer = 1*: Here, the hidden layer is represented as $\mathbf{h}_1^{(t)}$ and it receives its input from the data point, $\mathbf{x}^{(t)}$, and the hidden values in the same layer, but at the previous time step, $\mathbf{h}_1^{(t-1)}$.
- *layer = 2*: The second hidden layer, $\mathbf{h}_2^{(t)}$, receives its inputs from the outputs of the layer below at the current time step ($\mathbf{o}_1^{(t)}$) and its own hidden values from the previous time step, $\mathbf{h}_2^{(t-1)}$.

Since, in this case, each recurrent layer must receive a sequence as input, all the recurrent layers except the last one must return a sequence as output (that is, `return_sequences=True`). The behavior of the last recurrent layer depends on the type of problem.

Computing activations in an RNN

Each directed edge (the connections between boxes) in the representation of an RNN that we just looked at is associated with a weight matrix. Those weights do not depend on time, t ; therefore, they are shared across the time axis. The different weight matrices in a single-layer RNN are as follows:

- $\mathbf{W}_{xh}$: The weight matrix between the input, $\mathbf{x}^{(t)}$, and the hidden layer, h
- $\mathbf{W}_{hh}$: The weight matrix associated with the recurrent edge
- $\mathbf{W}_{ho}$: The weight matrix between the hidden layer and output layer

These weight matrices are depicted in the following figure:

```
Image(filename='images/16_05.png', width=700)
```

Computing the activations is very similar to standard multilayer perceptrons and other types of feedforward NNs. For the hidden layer, the net input, $\mathbf{z}_h$ (preactivation), is computed through a linear combination, that is, we compute the sum of the multiplications of the weight matrices with the corresponding vectors and add the bias unit:

$$\mathbf{z}_h^{(t)} = \mathbf{W}_{xh} \mathbf{x}_{(t)} + \mathbf{W}_{hh} \mathbf{h}^{(t-1)} + \mathbf{b}^h$$

Then, the activations of the hidden units at the time step, t , are calculated as follows:

$$\mathbf{h}^{(t)} = \phi_h(\mathbf{z}_h^{(t)}) = \phi_h(\mathbf{W}_{xh} \mathbf{x}_{(t)} + \mathbf{W}_{hh} \mathbf{h}^{(t-1)} + \mathbf{b}^h)$$

Here, $\mathbf{b}^h$ is the bias vector for the hidden units and $\phi_h(\cdot)$ is the activation function of the hidden layer.

Once the activations of the hidden units at the current time step are computed, the activations of the output units will be computed, as follows:

$$\mathbf{o}^{(t)} = \phi_o(\mathbf{W}_{ho}\mathbf{h}^{(t)} + \mathbf{b}_o)$$

To help clarify this further, the following figure shows the process of computing these activations with both formulations:

```
Image(filename='images/16_06.png', width=700)
```

Implementing RNNs for sequence modeling in TensorFlow.

Predicting the sentiment of IMDb movie reviews

We will implement a many-to-many RNN for an application of language modeling. While the chosen examples are purposefully simple to introduce the main concepts of RNNs, language modeling has a wide range of interesting applications, such as building chatbots—giving computers the ability to directly talk and interact with humans.

Preparing the movie review data

We will use a clean dataset named `movie_data.csv`. First, we will import the necessary modules and read the data into a pandas `DataFrame`, as follows:

```
import tensorflow as tf
import tensorflow_datasets as tfds
import numpy as np
import pandas as pd

df = pd.read_csv('movie_data.csv', encoding='utf-8')

df.tail()
```

Remember that this data frame, `df`, consists of two columns, namely `'review'` and `'sentiment'`, where `'review'` contains the text of movie reviews (the input features), and `'sentiment'` represents the target label we want to predict (0 refers to negative sentiment and 1 refers to positive sentiment). The text component of these movie reviews is sequences of words, and the RNN model classifies each sequence as a positive (1) or negative (0) review. However, before we can feed the data into an RNN model, we need to apply several preprocessing steps:

1. Create a `TensorFlow` dataset object and split it into separate training, testing, and validation partitions.
2. Identify the unique words in the training dataset.
3. Map each unique word to a unique integer and encode the review text into encoded integers (an index of each unique word).
4. Divide the dataset into mini-batches as input to the model.

```
# Step 1: Create a dataset
```

```
target = df.pop('sentiment')
```

```
ds_raw = tf.data.Dataset.from_tensor_slices(
    (df.values, target.values))

## inspection:
for ex in ds_raw.take(3):
    tf.print(ex[0].numpy()[0][:50], ex[1])
```

- **Train/validation/test splits** Now, we can split it into training, testing, and validation datasets. The entire dataset contains 50,000 examples. We will keep the first 25,000 examples for evaluation (hold-out testing dataset), and then 20,000 examples will be used for training and 5,000 for validation. The code is as follows:

```
tf.random.set_seed(1)

ds_raw = ds_raw.shuffle(
    50000, reshuffle_each_iteration=False)

ds_raw_test = ds_raw.take(25000)
ds_raw_train_valid = ds_raw.skip(25000)
ds_raw_train = ds_raw_train_valid.take(20000)
ds_raw_valid = ds_raw_train_valid.skip(20000)
```

- **Tokenizer and Encoder**

To prepare the data for input to a NN, we need to encode it into numeric values, as was mentioned in steps 2 and 3. To do this, we will first find the unique words (tokens) in the training dataset. While finding unique tokens is a process for which we can use Python datasets, it can be more efficient to use the `Counter` class from the `collections` package, which is part of Python's standard library. In the following code, we will instantiate a new `Counter` object (`token_counts`) that will collect the unique word frequencies. Note that in this particular application (and in contrast to the bag-of-words model), we are only interested in the set of unique words and won't require the word counts, which are created as a side product. To split the text into words (or tokens), the `tensorflow_datasets` package provides a `Tokenizer` class.

- `tfds.features.text.Tokenizer`:
https://www.tensorflow.org/datasets/api_docs/python/tfds/features/text/Tokenizer
- `tfds.features.text.TokenTextEncoder`:
https://www.tensorflow.org/datasets/api_docs/python/tfds/features/text/TokenTextEncoder
- **Encoding sequences: keeping the last 100 items in each sequence**

The code for collecting unique tokens is as follows:

```
## Step 2: find unique tokens (words)

from collections import Counter

try:
    tokenizer = tfds.features.text.Tokenizer()
except AttributeError:
    tokenizer = tfds.deprecated.text.Tokenizer()
```

```
token_counts = Counter()

for example in ds_raw_train:
    tokens = tokenizer.tokenize(example[0].numpy()[0])
    token_counts.update(tokens)

print('Vocab-size:', len(token_counts))
```

Next, we are going to map each unique word to a unique integer. This can be done manually using a Python dictionary, where the keys are the unique tokens (words) and the value associated with each key is a unique integer. However, the `tensorflow_datasets` package already provides a class, `TokenTextEncoder`, which we can use to create such a mapping and encode the entire dataset. First, we will create an encoder object from the `TokenTextEncoder` class by passing the unique tokens (`token_counts` contains the tokens and their counts, although here, their counts are not needed, so they will be ignored). Calling the `encoder.encode()` method will then convert its input text into a list of integer values:

```
## Step 3: encoding each unique token into integers

try:
    encoder = tfds.features.text.TokenTextEncoder(token_counts)
except AttributeError:
    encoder = tfds.deprecated.text.TokenTextEncoder(token_counts)

example_str = 'This is an example!'
encoder.encode(example_str)
```

Note that there might be some tokens in the validation or testing data that are not present in the training data and are thus not included in the mapping. If we have q tokens (that is the size of `token_counts` passed to the `TokenTextEncoder`, which in this case is 87,007), then all tokens that haven't been seen before, and are thus not included in `token_counts`, will be assigned the integer $q+1$ (which will be 87,008 in our case). In other words, the index $q+1$ is reserved for unknown words. Another reserved value is the integer 0, which serves as a placeholder for adjusting the sequence length.

We can use the `map()` method of the dataset objects to transform each text in the dataset accordingly, just like we would apply any other transformation to a dataset. However, there is a small problem: here, the text data is enclosed in tensor objects, which we can access by calling the `numpy()` method on a tensor in the eager execution mode. But during transformations by the `map()` method, the eager execution will be disabled. To solve this problem, we can define two functions. The first function will treat the input tensors as if the eager execution mode is enabled:

In the second function, we will wrap the first function using `tf.py_function` to convert it into a TensorFlow operator, which can then be used via its `map()` method. This process of encoding text into a list of integers can be carried out using the following code:

```
## Step 3-A: define the function for transformation

def encode(text_tensor, label):
    text = text_tensor.numpy()[0]
    encoded_text = encoder.encode(text)
```

```

        return encoded_text, label

## Step 3-B: wrap the encode function to a TF Op.
def encode_map_fn(text, label):
    return tf.py_function(encode, inp=[text, label],
                          Tout=(tf.int64, tf.int64))

ds_train = ds_raw_train.map(encode_map_fn)
ds_valid = ds_raw_valid.map(encode_map_fn)
ds_test = ds_raw_test.map(encode_map_fn)

tf.random.set_seed(1)
for example in ds_train.shuffle(1000).take(5):
    print('Sequence length:', example[0].shape)

example

```

So far, we've converted sequences of words into sequences of integers. However, there is one issue that we still need to resolve—the sequences currently have different lengths (as shown in the result of executing the previous code for five randomly chosen examples). Although, in general, RNNs can handle sequences with different lengths, we still need to make sure that all the sequences in a mini-batch have the same length to store them efficiently in a tensor.

- **batch() vs. padded_batch()**

To divide a dataset that has elements with different shapes into mini-batches, TensorFlow provides a different method, `padded_batch()` (instead of `batch()`), which will automatically pad the consecutive elements that are to be combined into a batch with placeholder values (0s) so that all sequences within a batch will have the same shape. To illustrate this with a practical example, let's take a small subset of size 8 from the training dataset, `ds_train`, and apply the `padded_batch()` method to this subset with `batch_size=4`. We will also print the sizes of the individual elements before combining these into mini-batches, as well as the dimensions of the resulting mini-batches:

```

# this will result in error

BATCH_SIZE = 32
train_data = all_encoded_data.batch(BATCH_SIZE)

next(iter(train_data))

# Running this will result in error
# We cannot apply .batch() to this dataset

## Take a small subset

ds_subset = ds_train.take(8)
for example in ds_subset:
    print('Individual size:', example[0].shape)

## batching the datasets

```



```
ds_batched = ds_subset.padded_batch(
    4, padded_shapes=([-1], []))

for batch in ds_batched:
    print('Batch dimension:', batch[0].shape)
```

As you can observe from the printed tensor shapes, the number of columns (that is, `.shape[1]`) in the first batch is 688, which resulted from combining the first four examples into a single batch and using the maximum size of these examples. That means that the other three examples in this batch are padded as much as necessary to match this size. Similarly, the second batch keeps the maximum size of its individual four examples, which is 453, and pads the other examples so that their length is smaller than the maximum length. Let's divide all three datasets into mini-batches with a batch size of 32:

```
## batching the datasets
train_data = ds_train.padded_batch(
    32, padded_shapes=([-1], []))

valid_data = ds_valid.padded_batch(
    32, padded_shapes=([-1], []))

test_data = ds_test.padded_batch(
    32, padded_shapes=([-1], []))
```

Now, the data is in a suitable format for an RNN model, which we are going to implement in the following subsections. However, we will first discuss feature embedding, which is an optional but highly recommended preprocessing step that is used for reducing the dimensionality of the word vectors.

Embedding layers for sentence encoding

During the data preparation in the previous step, we generated sequences of the same length. The elements of these sequences were integer numbers that corresponded to the indices of unique words. These word indices can be converted into input features in several different ways. One naive way is to apply one-hot encoding to convert the indices into vectors of zeros and ones. Then, each word will be mapped to a vector whose size is the number of unique words in the entire dataset. Given that the number of unique words (the size of the vocabulary) can be in the order of $10^4 - 10^5$, which will also be the number of our input features, a model trained on such features may suffer from the curse of dimensionality. Furthermore, these features are very sparse, since all are zero except one.

A more elegant approach is to map each word to a vector of a fixed size with realvalued elements (not necessarily integers). In contrast to the one-hot encoded vectors, we can use finite-sized vectors to represent an infinite number of real numbers. (In theory, we can extract infinite real numbers from a given interval, for example $[-1, 1]$). This is the idea behind embedding, which is a feature-learning technique that we can utilize here to automatically learn the salient features to represent the words in our dataset. Given the number of unique words, n_{words} , we can select the size of the embedding vectors (a.k.a., embedding dimension) to be much smaller than the number of unique words ($embedding_{dim} \ll n_{words}$) to represent the entire vocabulary as input features. The advantages of embedding over one-hot encoding are as follows:

- A reduction in the dimensionality of the feature space to decrease the effect of the curse of dimensionality
- The extraction of salient features since the embedding layer in an NN can be optimized (or learned)
- `input_dim`: number of words, i.e. maximum integer index + 1.
- `output_dim`:
- `input_length`: the length of (padded) sequence
 - for example, 'This is an example' -> [0, 0, 0, 0, 0, 0, 3, 1, 8, 9]
=> `input_length` is 10
- When calling the layer, takes integer values as input, the embedding layer convert each integer into float vector of size `[output_dim]`
 - If input shape is `[BATCH_SIZE]`, output shape will be `[BATCH_SIZE, output_dim]`
 - If input shape is `[BATCH_SIZE, 10]`, output shape will be `[BATCH_SIZE, 10, output_dim]`

The following schematic representation shows how embedding works by mapping token indices to a trainable embedding matrix:

```
Image(filename='images/16_10.png', width=700)
```

Given a set of tokens of size $n+2$ (n is the size of the token set, plus index 0 is reserved for the padding placeholder, and $n+1$ is for the words not present in the token set), an embedding matrix of size $(n+2) \times embedding_{dim}$ will be created where each row of this matrix represents numeric features associated with a token. Therefore, when an integer index, i , is given as input to the embedding, it will look up the corresponding row of the matrix at index i and return the numeric features. The embedding matrix serves as the input layer to our NN models.

In practice, creating an embedding layer can simply be done using `tf.keras.layers.Embedding`. Let's see an example where we will create a model and add an embedding layer, as follows:

```
from tensorflow import keras
from keras import layers
from keras.layers import Embedding

model = tf.keras.Sequential()

model.add(Embedding(input_dim=100,
                    output_dim=6,
                    input_length=20,
                    name='embed-layer'))

model.summary()
```

Building an RNN model

Now we're ready to build an RNN model. Using the Keras `Sequential` class, we can combine the embedding layer, the recurrent layers of the RNN, and the fully connected non-recurrent layers. For the recurrent layers, we can use any of the following implementations:

- *SimpleRNN*: a regular RNN layer, that is, a fully connected recurrent layer
- *LSTM*: a long short-term memory RNN, which is useful for capturing the long-term dependencies
- *GRU*: a recurrent layer with a gating RNN Encoder-Decoder for [Statistical Machine Translation](#), as an alternative to LSTMs To see how a multilayer RNN model can be built using one of these recurrent layers, in the following example, we will create an RNN model, starting with an embedding layer with `input_dim=1000` and `output_dim=32`. Then, two recurrent layers of type *SimpleRNN* will be added. Finally, we will add a non-recurrent fully connected layer as the output layer, which will return a single output value as the prediction.
- **Keras RNN layers:**
 - `tf.keras.layers.SimpleRNN(units, return_sequences=False)`
 - `tf.keras.layers.LSTM(..)`
 - `tf.keras.layers.GRU(..)`
 - `tf.keras.layers.Bidirectional()`
- **Determine `return_sequences=?`**
 - In a multi-layer RNN, all RNN layers except the last one should have `return_sequences=True`
 - For the last RNN layer, decide based on the type of problem:
 - many-to-many: -> `return_sequences=True`
 - many-to-one: -> `return_sequences=False`
 - ..

```
## An example of building a RNN model
## with SimpleRNN layer
```

```
from tensorflow import keras
from keras import layers, Sequential
from keras.layers import Embedding
from keras.layers import SimpleRNN
from keras.layers import Dense

model = Sequential()
model.add(Embedding(1000, 32))
model.add(SimpleRNN(32, return_sequences=True))
model.add(SimpleRNN(32))
model.add(Dense(1))
model.summary()
```

```
## An example of building a RNN model
## with LSTM layer
```

```

from keras.layers import LSTM

model = Sequential()
model.add(Embedding(10000, 32))
model.add(LSTM(32, return_sequences=True))
model.add(LSTM(32))
model.add(Dense(1))
model.summary()

## An example of building a RNN model
## with GRU layer
from keras.layers import GRU

model = Sequential()
model.add(Embedding(10000, 32))
model.add(GRU(32, return_sequences=True))
model.add(GRU(32))
model.add(Dense(1))
model.summary()

```

Building an RNN model for the sentiment analysis task

Since we have very long sequences, we are going to use an LSTM layer to account for long-term effects. In addition, we will put the LSTM layer inside a Bidirectional wrapper, which will make the recurrent layers pass through the input sequences from both directions, start to end, as well as the reverse direction:

```

embedding_dim = 20
vocab_size = len(token_counts) + 2

tf.random.set_seed(1)

## build the model
bi_lstm_model = tf.keras.Sequential([
    tf.keras.layers.Embedding(
        input_dim=vocab_size,
        output_dim=embedding_dim,
        name='embed-layer'),

    tf.keras.layers.Bidirectional(
        tf.keras.layers.LSTM(64, name='lstm-layer'),
        name='bidir-lstm'),

    tf.keras.layers.Dense(64, activation='relu'),

    tf.keras.layers.Dense(1, activation='sigmoid')
])

bi_lstm_model.summary()

## compile and train:

```

```

bi_lstm_model.compile(
    optimizer=tf.keras.optimizers.Adam(1e-3),
    loss=tf.keras.losses.BinaryCrossentropy(from_logits=False),
    metrics=['accuracy'])

history = bi_lstm_model.fit(
    train_data,
    validation_data=valid_data,
    epochs=10)

## evaluate on the test data
test_results= bi_lstm_model.evaluate(test_data)
print('Test Acc.: {:.2f}%'.format(test_results[1]*100))

```

After training this model for 10 epochs, evaluation on the test data shows 85 percent accuracy. (Note that this result is not the best when compared to the state-of-the-art methods used on the IMDb dataset. The goal was simply to show how RNN works.)

```

import os

if not os.path.exists('models'):
    os.mkdir('models')

bi_lstm_model.save('models/Bidir-LSTM-full-length-seq.h5')

```

- **Trying SimpleRNN with short sequences**

In order to obtain reasonable predictive performance on this dataset using a SimpleRNN, we can truncate the sequences. Also, utilizing our "domain knowledge," we may hypothesize that the last paragraphs of a movie review may contain most of the information about its sentiment. Hence, we can focus only on the last portion of each review. To do this, we will define a helper function, `preprocess_datasets()`, to combine the preprocessing steps 2-4. An optional argument to this function is `max_seq_length`, which determines how many tokens from each review should be used. For example, if we set `max_seq_length=100` and a review has more than 100 tokens, only the last 100 tokens will be used. If `max_seq_length` is set to `None`, then full-length sequences will be used as before. Trying different values for `max_seq_length` will give us more insights on the capability of different RNN models to handle long sequences.

The code for the `preprocess_datasets()` function is as follows:

```

def preprocess_datasets(
    ds_raw_train,
    ds_raw_valid,
    ds_raw_test,
    max_seq_length=None,
    batch_size=32):

    ## Step 1: (already done => creating a dataset)
    ## Step 2: find unique tokens
    tokenizer = tfds.features.text.Tokenizer()
    token_counts = Counter()

```

```

for example in ds_raw_train:
    tokens = tokenizer.tokenize(example[0].numpy()[0])
    if max_seq_length is not None:
        tokens = tokens[-max_seq_length:]
    token_counts.update(tokens)

print('Vocab-size:', len(token_counts))

## Step 3: encoding the texts
encoder = tfds.features.text.TokenTextEncoder(token_counts)
def encode(text_tensor, label):
    text = text_tensor.numpy()[0]
    encoded_text = encoder.encode(text)
    if max_seq_length is not None:
        encoded_text = encoded_text[-max_seq_length:]
    return encoded_text, label

def encode_map_fn(text, label):
    return tf.py_function(encode, inp=[text, label],
                          Tout=(tf.int64, tf.int64))

ds_train = ds_raw_train.map(encode_map_fn)
ds_valid = ds_raw_valid.map(encode_map_fn)
ds_test = ds_raw_test.map(encode_map_fn)

## Step 4: batching the datasets
train_data = ds_train.padded_batch(
    batch_size, padded_shapes=([-1],[]))

valid_data = ds_valid.padded_batch(
    batch_size, padded_shapes=([-1],[]))

test_data = ds_test.padded_batch(
    batch_size, padded_shapes=([-1],[]))

return (train_data, valid_data,
        test_data, len(token_counts))

```

Next, we will define another helper function, `build_rnn_model()`, for building models with different architectures more conveniently:

```

def build_rnn_model(embedding_dim, vocab_size,
                    recurrent_type='SimpleRNN',
                    n_recurrent_units=64,
                    n_recurrent_layers=1,
                    bidirectional=True):

    tf.random.set_seed(1)

    # build the model
    model = tf.keras.Sequential()

```

```

model.add(
    Embedding(
        input_dim=vocab_size,
        output_dim=embedding_dim,
        name='embed-layer')
)

for i in range(n_recurrent_layers):
    return_sequences = (i < n_recurrent_layers-1)

    if recurrent_type == 'SimpleRNN':
        recurrent_layer = SimpleRNN(
            units=n_recurrent_units,
            return_sequences=return_sequences,
            name='simprnn-layer-{}'.format(i))
    elif recurrent_type == 'LSTM':
        recurrent_layer = LSTM(
            units=n_recurrent_units,
            return_sequences=return_sequences,
            name='lstm-layer-{}'.format(i))
    elif recurrent_type == 'GRU':
        recurrent_layer = GRU(
            units=n_recurrent_units,
            return_sequences=return_sequences,
            name='gru-layer-{}'.format(i))

    if bidirectional:
        recurrent_layer = Bidirectional(
            recurrent_layer, name='bidir-'+recurrent_layer.name)

    model.add(recurrent_layer)

model.add(tf.keras.layers.Dense(64, activation='relu'))
model.add(tf.keras.layers.Dense(1, activation='sigmoid'))

return model

```

Now, using these two fairly general, but convenient, helper functions, we can readily compare different RNN models with different input sequence lengths. As an example, in the following code, we will try a model with a single recurrent layer of type SimpleRNN while truncating the sequences to a maximum length of 100 tokens:

```

from keras.layers import Bidirectional

batch_size = 32
embedding_dim = 20
max_seq_length = 100

train_data, valid_data, test_data, n = preprocess_datasets(
    ds_raw_train, ds_raw_valid, ds_raw_test,
    max_seq_length=max_seq_length,
    batch_size=batch_size
)

```

```

vocab_size = n + 2

rnn_model = build_rnn_model(
    embedding_dim, vocab_size,
    recurrent_type='SimpleRNN',
    n_recurrent_units=64,
    n_recurrent_layers=1,
    bidirectional=True)

rnn_model.summary()

rnn_model.compile(optimizer=tf.keras.optimizers.Adam(1e-3),
    loss=tf.keras.losses.BinaryCrossentropy(from_logits=False),
    metrics=['accuracy'])

history = rnn_model.fit(
    train_data,
    validation_data=valid_data,
    epochs=10)

results = rnn_model.evaluate(test_data)

print('Test Acc.: {:.2f}%'.format(results[1]*100))

```

For instance, truncating the sequences to 100 tokens and using a bidirectional SimpleRNN layer resulted in 80 percent classification accuracy. Although the prediction is slightly lower when compared to the previous bidirectional LSTM model (85.15 percent accuracy on the test dataset), the performance on these truncated sequences is much better than the performance we could achieve with a SimpleRNN on full-length movie reviews. As an optional exercise, you can verify this by using the two helper functions we have already defined. Try it with `max_seq_length=None` and set the `bidirectional` argument inside the `build_rnn_model()` helper function to `False`

3. Real-world Examples

Neural networks find extensive use in various real-world applications, including the realm of autonomous vehicles. In this context, they enable vehicles to perceive their surroundings, make informed decisions, and navigate complex traffic scenarios, thereby enhancing road safety and efficiency. Moreover, in the field of medical diagnosis, neural networks play a pivotal role in interpreting medical images, aiding in the early detection of diseases such as cancer, and facilitating more accurate diagnoses, leading to improved treatment outcomes and patient care. Furthermore, industries such as finance leverage neural networks to detect fraud patterns, assess risks, and make data-driven investment decisions, contributing to enhanced security and optimized financial strategies.

Addressing bias and fairness in neural network models is crucial to ensure equitable outcomes and prevent discrimination. Several approaches, including data preprocessing techniques, algorithmic modifications, and the inclusion of diverse datasets, are employed to mitigate biases. Implementing techniques such as adversarial training helps neural networks learn to identify and rectify biased patterns, promoting fairness in decision-making

processes. Moreover, transparency in model design and decision-making algorithms is emphasized, enabling stakeholders to understand the factors influencing outcomes and detect any potential biases. Collaborative efforts involving multidisciplinary teams are instrumental in developing comprehensive frameworks that prioritize fairness and equity, fostering a more inclusive and just technological landscape. Additionally, ongoing research into interpretable AI and explainable neural network models plays a pivotal role in enhancing transparency, enabling users to comprehend the reasoning behind the model's decisions and identify any biases embedded in the system.

Neural Networks. Practical Tasks

Task 1. Implement a Perceptron from Scratch

Problem: Creating a Perceptron Model for XOR Classification

Task: Develop a simple perceptron model using Python and NumPy to perform binary classification on the XOR problem.

Description: The XOR problem involves classifying a set of input pairs (x_1, x_2) into two classes: 1 if the inputs are different (0, 1 or 1, 0), and 0 if the inputs are the same (0, 0 or 1, 1). A perceptron is a simple neural network model that can learn linear decision boundaries. In this task, you will create a perceptron, train it on the XOR dataset, and observe its classification performance.

Steps:

1. Implement a Perceptron class with methods for initialization, activation, prediction, and training.
2. Initialize the perceptron's weights and bias with random values.
3. Define the XOR dataset: input features and corresponding labels.
4. Use the perceptron's `train` method to update weights and bias using the perceptron learning rule.
5. Test the trained perceptron on the XOR dataset and print its predictions.

Expected Output: For each input pair (x_1, x_2) in the XOR dataset, the perceptron should predict the corresponding class label (0 or 1). The final output should demonstrate the perceptron's ability to learn and classify the XOR problem.

```
import numpy as np

class Perceptron:
    def __init__(self, input_size):
        """
        Initializes a simple perceptron model.

        Args:
            input_size (int): Number of input features.
        """

    def activation(self, x):
        """
        Activation function (Step function).

        Args:
            x (float): Input value.

        Returns:
            int: 1 if x >= 0, else 0.
```

```

        """
        return result

    def predict(self, x):
        """
        Predicts the output label using the perceptron model.

        Args:
            x (ndarray): Input features.

        Returns:
            int: Predicted label (1 or 0).
        """

        return result

    def train(self, X, y, num_epochs, learning_rate):
        """
        Trains the perceptron model on the given dataset using the
        perceptron learning rule.

        Args:
            X (ndarray): Input features of the dataset.
            y (ndarray): Ground truth labels of the dataset.
            num_epochs (int): Number of training epochs.
            learning_rate (float): Learning rate for weight update.
        """

        return None

# XOR dataset: Input features and corresponding labels
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 1, 1, 0])

# Create and train the perceptron model
perceptron = Perceptron(input_size=2)
perceptron.train(X, y, num_epochs=1000, learning_rate=0.1)

# Test the trained model
test_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
for data in test_data:
    prediction = perceptron.predict(data)
    print(f"Input: {data}, Prediction: {prediction}")

```

Task 2. Creating a visualization of common activation functions (sigmoid, ReLU, and tanh) using Python and Matplotlib

Problem: Activation Function Visualization

Task: Create a Visualization of Common Activation Functions

Description: Activation functions are crucial components in neural networks that introduce non-linearity and enable the network to learn complex patterns. This task involves visualizing the behavior of common activation functions: sigmoid, ReLU (Rectified Linear Unit), and tanh (hyperbolic tangent). By plotting their graphs, you will gain insights into how they transform input values.

Steps:

1. Implement the activation functions `sigmoid`, `relu`, and `tanh`.
2. Generate a range of input values for plotting.
3. Calculate the output values for each activation function using the provided input range.
4. Use Matplotlib to create a plot that shows the graphs of all three activation functions on the same plot.
5. Add labels, title, and legend to the plot to make it informative.

Expected Output: A single plot should display the graphs of the sigmoid, ReLU, and tanh activation functions. The x-axis should represent the input values, and the y-axis should represent the output values. Each activation function's graph should have a distinctive shape that showcases its unique characteristics.

Tips:

- Sigmoid: S-shaped curve that maps input values to a range between 0 and 1.
- ReLU: Linear for positive values and zero for negative values, creating a hinge-like shape.
- tanh: S-shaped curve that maps input values to a range between -1 and 1.

Learning Objectives: This task will help you understand the effects of different activation functions on input data transformation. You'll observe how these functions introduce non-linearity, saturate, or allow only positive outputs, which are essential aspects of neural network behavior.

```
import numpy as np
import matplotlib.pyplot as plt

def sigmoid(x):
    """
    Sigmoid activation function.

    Args:
        x (float): Input value.

    Returns:
        float: Output of the sigmoid function.
    """
    return result

def relu(x):
    """
    Rectified Linear Unit (ReLU) activation function.

    Args:
        x (float): Input value.
```

```

    Returns:
        float: Output of the ReLU function.
    """
    return result

def tanh(x):
    """
    Hyperbolic tangent (tanh) activation function.

    Args:
        x (float): Input value.

    Returns:
        float: Output of the tanh function.
    """
    return result

# Create x values for plotting

# Calculate y values for each activation function

# Plot the activation functions

# Add labels and title

# Show the plot

```

Task 3. Hyperparameter Tuning

Problem: Hyperparameter Tuning for Neural Network

Task: Experiment with Different Hyperparameters to Optimize Neural Network Performance

Description: Hyperparameters are crucial settings that impact the performance of neural network models. This task involves exploring the effects of different hyperparameters, such as learning rate and number of hidden units, on the performance of a neural network model. By conducting experiments with various hyperparameter values, you will gain insights into how these settings influence the training process and final results.

Steps:

1. Implement a function to create a neural network model with a specified number of hidden units.
2. Implement a function to train the neural network model on the provided training data.
3. Define the hyperparameters to experiment with: learning rates and hidden units.
4. Loop through different combinations of hyperparameters and perform the following:
 - Create a new neural network model with the specified hyperparameters.
 - Train the model using the training data and provided hyperparameters.

- Print the results of each experiment, including learning rate, hidden units, and training performance.

Expected Output: For each combination of hyperparameters, you will observe the learning process of the neural network as it trains on the provided data. The output will include training progress updates and performance metrics such as loss and accuracy.

Tips:

- Experiment with a range of learning rates (e.g., 0.001, 0.01, 0.1) and hidden units (e.g., 4, 8, 16).
- Observe how different learning rates affect convergence speed and final performance.
- Notice the impact of changing the number of hidden units on model complexity and overfitting.

Learning Objectives: This task will help you understand the significance of hyperparameters in neural networks and their effects on model training and performance. You will gain hands-on experience in tuning hyperparameters and making informed decisions to optimize your neural network models.

Note. In code, we define two functions:

- `create_neural_network`: Creates a neural network model with a specified number of hidden units and compiles it using the Adam optimizer.
- `train_neural_network`: Trains the given neural network model on the provided training data.

```
!pip install keras

import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD

def create_neural_network(input_dim, hidden_units):
    """
    Create a neural network model with a specified number of hidden
    units.

    Args:
        input_dim (int): Number of input features.
        hidden_units (int): Number of units in the hidden layer.

    Returns:
        model (Sequential): Compiled neural network model.
    """

    return model

def train_neural_network(model, X_train, y_train, num_epochs,
batch_size):
    """
    Train the neural network model on the training data.
```

```

    Args:
        model (Sequential): Compiled neural network model.
        X_train (ndarray): Training input features.
        y_train (ndarray): Training labels.
        num_epochs (int): Number of training epochs.
        batch_size (int): Batch size for training.
    """
    return None

# Example data and labels

# Hyperparameters to experiment with

# Perform hyperparameter tuning

```

Task 4. Text Classification with Word Embeddings

Problem: Text Classification with Word Embeddings

Task: Train a Neural Network for Sentiment Analysis Using Word Embeddings

Description: Sentiment analysis is the process of determining the sentiment or emotional tone expressed in a piece of text. This task involves training a simple neural network to perform sentiment analysis on a dataset of text reviews. Word embeddings will be utilized to represent words in a dense vector space, capturing semantic relationships between words. The trained model will then predict whether a given text review has a positive or negative sentiment.

Steps:

1. Load and preprocess the dataset of text reviews and their corresponding sentiment labels (positive or negative).
2. Tokenize the text data to convert it into sequences of numerical values (word indices).
3. Pad the sequences to ensure uniform length for neural network input.
4. Build a neural network model with an embedding layer to learn word representations and a dense output layer for binary sentiment classification.
5. Train the model using the tokenized and padded text data, aiming to minimize the binary cross-entropy loss.
6. Evaluate the model's performance using evaluation metrics such as accuracy, precision, recall, and F1-score.

Expected Output: A trained neural network model capable of predicting the sentiment (positive or negative) of text reviews. This model should be able to process new text reviews and classify their sentiments accurately.

Tips:

- Use a well-known dataset for sentiment analysis, such as IMDb movie reviews.
- Experiment with different hyperparameters like embedding dimension, number of hidden layers, and batch size.
- Visualize the training progress by plotting the loss and accuracy curves over epochs.

Learning Objectives: By completing this task, you'll gain practical experience in natural language processing and sentiment analysis. You'll understand the importance of word embeddings in capturing semantic information from text and how neural networks can be effectively used for text classification tasks.

```
import numpy as np
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense
from keras.preprocessing.text import Tokenizer
from keras.utils import pad_sequences
from keras.models import load_model

def train_sentiment_analysis_model(texts, labels, max_words,
    embedding_dim, num_epochs, batch_size):
    """
    Train a neural network model for sentiment analysis using word
    embeddings.

    Args:
        texts (list): List of text reviews.
        labels (list): List of corresponding sentiment labels (0 or
    1).
        max_words (int): Maximum number of words to tokenize.
        embedding_dim (int): Dimension of word embeddings.
        num_epochs (int): Number of training epochs.
        batch_size (int): Batch size for training.
    """

    return None

# Example data: text reviews and sentiment labels

# Hyperparameters

# Train the sentiment analysis model

# Test the model

# Load the trained model
# Load your model file

# Example new text reviews

# Tokenize and pad the new text reviews

# Use the trained model to predict sentiments

# Convert the predictions to binary labels (0 or 1)

# Print the predicted sentiments for new text reviews
```


Topic: Natural Language Processing in AI

Duration: 2 hours

Agenda:

1. Introduction to Natural Language Processing (NLP)
 - a. Understanding the basics of Natural Language Processing
 - b. Importance of NLP in AI applications
 - c. Real-world examples of NLP in everyday life
2. Text Preprocessing
 - a. Cleaning and tokenization of text data
 - b. Stop word removal and stemming
 - c. Handling special characters and numbers
3. Bag-of-Words Model
 - a. Concept of the bag-of-words model
 - b. Representing text data as numerical vectors
 - c. Building a simple sentiment analysis model using the bag-of-words approach
4. Word Representation
 - a. Introduction to word embeddings (e.g., Word2Vec, GloVe)
 - b. Word vectors and semantic relationships
 - c. Applications of word embeddings in NLP tasks
5. Practical NLP Applications
6. Q&A and Wrap-up

##1. Introduction to Natural Language Processing (NLP)

Natural Language Processing (NLP) is a part of AI (artificial intelligence) that deals with understanding and processing of human language. In real time, majority of data exists in the unstructured form such as text, videos, images. Mass of data in unstructured category, will be in textual form. To process this textual data's with machine learning algorithms, NLP comes in to play.

NLP use cases are Language translation, Speech recognition, Hiring and Recruitment, Chat Bot, Sentimental analysis and so on.

Examples

1) Sentimental analysis

Sentimental analysis predicts whether the review is positive or negative.

Positive Review:

"Love the phone. the phone is sleek and smooth and beautiful I highly recommend the phone you will not regret getting this phone. "

Negative Review:

"The telephone headset is of poor quality , not understood that Dijon you when you call that sounds very bad, not recommending this article.
I wanted to return it but it is very difficult and expensive it back from Venezuela , since the dollars in my country are scarce and would consist more than the..."

2) Named entity recognition

It is the task of identifying and categorizing entities from the text.

Input:

Dhoni is one of the famous cricketer in India team.

Output:

Dhoni | Person

India | Country

2. Text Preprocessing

###Terms used in NLP **Corpus** refers to the collection of documents or text files. The tweet data from twitter is a *corpus*.

A text sample defined as **Documents**. Each tweet is a *document*.

Document comprises of **Sentences**. Each tweet has one or more *sentences*.

A text get divided in to smaller unit called **Tokens**. Each tweet sentence has one or more *tokens*.

NLP Techniques For Text

Natural Language Toolkit (NLTK) is the leading platform for building Python programs to work with human language data. It provides pre-defined functions for classification, tokenization, stemming, tagging and parsing.

Explore it here <https://www.nltk.org/>

How to Perform NLP?

The steps to perform preprocessing of data in NLP include:

Segmentation:

You first need to break the entire document down into its constituent sentences. You can do this by segmenting the article along with its punctuations like full stops and commas.

<https://view.genial.ly/64200cb07a7e9f0019a1bebe/interactive-image-basic-interactive-image>

```
#import spacy library
import spacy

#load core english library
nlp = spacy.load("en_core_web_sm")

#take string
doc = nlp("The sky is clear. The stars are twinkling at night.")
#to print sentences
for sent in doc.sents:
    print(sent)

The sky is clear.
The stars are twinkling at night.
```

Tokenizing:

For the algorithm to understand these sentences, you need to get the words in a sentence and explain them individually to our algorithm. So, you break down your sentence into its constituent words and store them. This is called tokenizing, and each word is called a token.

<https://view.genial.ly/642018897a7e9f0019a1f152/interactive-image-copy-nlp-f2>

```
from spacy.tokenizer import Tokenizer
from spacy.lang.en import English
nlp = English ()
tokenizer = Tokenizer(nlp.vocab)
tokens = tokenizer("The stars are twinkling at night")
for token in tokens:
    print (token)

The
stars
are
twinkling
at
night
```

Removing Stop Words:

You can make the learning process faster by getting rid of non-essential words, which add little meaning to our statement and are just there to make our statement sound more cohesive. Words such as *was*, *in*, *is*, *and*, *the*, are called **stop words** and can be removed.

<https://view.genial.ly/64202951270f7600120078ce/interactive-image-nlp-f4>

```
from spacy.lang.en.stop_words import STOP_WORDS
filtered_sentence = []

for word in tokens:
    if word.is_stop == False:
        filtered_sentence.append(word)
for word in filtered_sentence:
    print(word)

stars
twinkling
night
```

Stemming:

It is the process of obtaining the Word Stem of a word. Word Stem gives new words upon adding affixes to them

<https://view.genial.ly/642184a9b4c54c00194dd4c3/interactive-image-nlp-f5>

```
# Unfortunately, spaCy has no module for stemming.
# Consequently we use NLTK to show an example

import nltk
from nltk.stem import PorterStemmer
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
nltk.download('punkt')

ps = PorterStemmer()

sentence = "The stars are twinkling at night"
words = word_tokenize(sentence)

print("Stemming:")
print("=====")
for w in words:
    print(w, " : ", ps.stem(w))

[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data] Unzipping tokenizers/punkt.zip.

Stemming:
=====
The : the
stars : star
are : are
twinkling : twinkl
at : at
night : night
```

Lemmatization:

The process of obtaining the *Root Stem* of a word. Root Stem gives the new base form of a word that is present in the dictionary and from which the word is derived. You can also identify the base words for different words based on the tense, mood, gender, etc.

<https://view.genial.ly/6421ed59520cda001718eaa9/interactive-image-nlp-f6>

```
import spacy

nlp = spacy.load('en_core_web_sm')
doc = nlp("The stars are twinkling at night")

for token in doc:
    print(token.text + "-->" + token.lemma_)

The-->the
stars-->star
are-->be
twinkling-->twinkle
at-->at
night-->night
```

Part of Speech Tagging:

Now, you must explain the concept of nouns, verbs, articles, and other parts of speech to the machine by adding these tags to our words. This is called *part of*.

<https://view.genial.ly/6421f0c0b4c54c00194efd9e/interactive-image-nlp-f7>

```
import spacy

nlp = spacy.load('en_core_web_sm')
doc = nlp("The stars are twinkling at night")

for word in doc:
    print(word.text, word.pos_)

The DET
stars NOUN
are AUX
twinkling VERB
at ADP
night NOUN
```

Named Entity Tagging:

Next, introduce your machine to pop culture references and everyday names by flagging names of movies, important personalities or locations, etc that may occur in the document. You do this by classifying the words into subcategories. This helps you find any keywords in a sentence. The subcategories are person, location, monetary value, quantity, organization, movie. Named entity recognition (NER) can help answering many real-world questions, such as:

- Which companies were mentioned in the news article?
- Were specified products mentioned in complaints or reviews?

- Does the tweet contain the name of a person? Does the tweet contain this person's location?

SpaCy's named entity recognition has been trained on the OntoNotes 5 corpus and it supports the following entity types:

TYPE	DESCRIPTION
PERSON	People, including fictional.
NORP	Nationalities or religious or political groups.
FAC	Buildings, airports, highways, bridges, etc.
ORG	Companies, agencies, institutions, etc.
GPE	Countries, cities, states.
LOC	Non-GPE locations, mountain ranges, bodies of water.
PRODUCT	Objects, vehicles, foods, etc. (Not services.)
EVENT	Named hurricanes, battles, wars, sports events, etc.
WORK_OF_ART	Titles of books, songs, etc.
LAW	Named documents made into laws.
LANGUAGE	Any named language.
DATE	Absolute or relative dates or periods.
TIME	Times smaller than a day.
PERCENT	Percentage, including "%".
MONEY	Monetary values, including unit.
QUANTITY	Measurements, as of weight or distance.
ORDINAL	"first", "second", etc.
CARDINAL	Numerals that do not fall under another type.

Source: SpaCy

Entity

We are using the sentence,

"Twinkling Star Company has seen steady growth in revenue over the past few years. In 2022, the company's revenue was \$50 million, which grew to \$70 million in 2023, representing a 40% YoY growth. The company has a subscription-based business model, and it charges a monthly fee to access its platform. The company's revenue growth can be attributed to its increasing user base, which has grown from 500,000 users in 2021 to 700,000 users in 2022. The company's gross margin has also improved over the years. In 2022, the company had a gross margin of 50%, which improved to 55% in 2023. The improvement in the gross margin can be attributed to the company's focus on reducing its cost of sales. Twinkling Star's Company net income has also improved over the years. In 2022, the company had a net loss of \$5 million, which improved to a net income of \$2 million in 2023. The improvement in

the net income can be attributed to the company's focus on reducing its operating expenses."

One of the nice things about Spacy is that we only need to apply *nlp* once, the entire background pipeline will return the objects.

```
import spacy
from spacy import displacy
from collections import Counter
import en_core_web_sm

nlp = en_core_web_sm.load()

from pprint import pprint

doc = nlp("""Twinkling Star Company has seen steady growth in
revenue over the past few years. In 2022, the company's revenue was
$50 million, which grew to $70 million in 2023, representing a 40%
YoY growth. The company has a subscription-based business model, and
it charges a monthly fee to access its platform. The company's
revenue growth can be attributed to its increasing user base, which
has grown from 500,000 users in 2021 to 700,000 users in 2022.
The company's gross margin has also improved over the years. In
2022, the company had a gross margin of 50%, which improved to 55%
in 2023. The improvement in the gross margin can be attributed to
the company's focus on reducing its cost of sales.
Twinkling Star's Company net income has also improved over the
years. In 2022, the company had a net loss of $5 million, which
improved to a net income of $2 million in 2023. The improvement in
the net income can be attributed to the company's focus on reducing
its operating expenses.""")
pprint([(X.text, X.label_) for X in doc.ents])

[('Twinkling Star Company', 'ORG'),
 ('the past few years', 'DATE'),
 ('2022', 'DATE'),
 ('$50 million', 'MONEY'),
 ('$70 million', 'MONEY'),
 ('2023', 'DATE'),
 ('40%', 'PERCENT'),
 ('YoY', 'GPE'),
 ('monthly', 'DATE'),
 ('500,000', 'CARDINAL'),
 ('2021', 'DATE'),
 ('700,000', 'CARDINAL'),
 ('2022', 'DATE'),
 ('the years', 'DATE'),
 ('2022', 'DATE'),
 ('50%', 'PERCENT'),
 ('55%', 'PERCENT'),
 ('2023', 'DATE'),
 ('Twinkling Star's Company', 'ORG'),
 ('the years', 'DATE'),
```

```
( '2022', 'DATE'),
( '$5 million', 'MONEY'),
( '$2 million', 'MONEY'),
( '2023', 'DATE')]
```

Token

During the above example, we were working on entity level, in the following example, we are demonstrating token-level entity annotation using the BILUO tagging scheme to describe the entity boundaries.

TAG	DESCRIPTION
BEGIN	The first token of a multi-token entity.
IN	An inner token of a multi-token entity.
LAST	The final token of a multi-token entity.
UNIT	A single-token entity.
OUT	A non-entity token.

Source: *SpaCy*

```
pprint([(X, X.ent_iob_, X.ent_type_) for X in doc])
```

Extracting named entity from an article

Now let's get serious with SpaCy and extracting named entities from a *Medium* article [Why do Stars Twinkle and Planets Don't?](#)

```
from bs4 import BeautifulSoup
import requests
import re

def url_to_string(url):
    res = requests.get(url)
    html = res.text
    soup = BeautifulSoup(html, 'html5lib')
    for script in soup(["script", "style", 'aside']):
        script.extract()
    return " ".join(re.split(r'[\n\t]+', soup.get_text()))

# Why do Stars Twinkle and Planets Don't?
ny_bb = url_to_string('https://en.wikipedia.org/wiki/Twinkling')
```



```
article = nlp(ny_bb)
len(article.ents)
```

113

There are 113 entities in the article and they are represented as 9 unique labels:

```
labels = [x.label_ for x in article.ents]
Counter(labels)
```

```
Counter({'ORG': 27,
        'PRODUCT': 3,
        'PERSON': 30,
        'DATE': 14,
        'CARDINAL': 15,
        'LANGUAGE': 1,
        'GPE': 8,
        'LOC': 7,
        'TIME': 2,
        'WORK_OF_ART': 3,
        'NORP': 3})
```

The following are three most frequent tokens.

```
items = [x.text for x in article.ents]
Counter(items).most_common(3)

[('Earth', 5), ('2', 3), ('Wikipedia', 2)]
```

Let's randomly select one sentence to learn more.

```
sentences = [x for x in article.sents]
print(sentences[10])
```

Most scintillation effects are caused by anomalous atmospheric refraction caused by small-scale fluctuations in air density usually related to temperature gradients.[3][4]

Let's run *displacy.render* to generate the raw markup.

```
displacy.render(nlp(str(sentences[10])), jupyter=True, style='ent')
<IPython.core.display.HTML object>
```

Using spaCy's built-in *displaCy visualizer*, here's what the above sentence and its dependencies look like:

```
displacy.render(nlp(str(sentences[10])), style='dep', jupyter =
True, options = {'distance': 120})
<IPython.core.display.HTML object>
```

```
[(x.orth_, x.pos_, x.lemma_) for x in [y
                                     for y
                                     in nlp(str(sentences[10]))
                                     if not y.is_stop and y.pos_ !=
                                     'PUNCT']]

[('scintillation', 'NOUN', 'scintillation'),
 ('effects', 'NOUN', 'effect'),
 ('caused', 'VERB', 'cause'),
 ('anomalous', 'ADJ', 'anomalous'),
 ('atmospheric', 'ADJ', 'atmospheric'),
 ('refraction', 'NOUN', 'refraction'),
 ('caused', 'VERB', 'cause'),
 ('small', 'ADJ', 'small'),
 ('scale', 'NOUN', 'scale'),
 ('fluctuations', 'NOUN', 'fluctuation'),
 ('air', 'NOUN', 'air'),
 ('density', 'NOUN', 'density'),
 ('usually', 'ADV', 'usually'),
 ('related', 'VERB', 'relate'),
 ('temperature', 'NOUN', 'temperature'),
 ('gradients.[3][4]', 'PROPN', 'gradients.[3][4'])]
```

Next, we verbatim, extract part-of-speech and lemmatize this sentence.

```
dict([(str(x), x.label_) for x in nlp(str(sentences[10])).ents])
print([(x, x.ent_iob_, x.ent_type_) for x in sentences[10]])
```

Finally, we visualize the entity of the entire article.

```
displacy.render(nlp(ny_bb), jupyter=True, style='ent')
```

After performing the preprocessing steps, you then give your resultant data to a machine learning algorithm like Naive Bayes, etc., to create your NLP application.

Check your [knowledge](#) about *NLP* ☞

##3. Bag-of-Words Model

Bag-of-words is a model that represents text as an unordered collection of words. This model ignores syntax and considers only the meanings of the words in the sentence. This approach is helpful in some classification tasks, such as sentiment analysis (another classification task would be distinguishing regular email from spam email). Sentiment analysis can be used, for instance, in product reviews, categorizing reviews as positive or negative. Consider the following sentences:

- "My grandson loved it! So much fun!"
- "Product broke after a few days."
- "One of the best games I've played in a long time."
- "Kind of cheap and flimsy, not worth it."

Based only on the words in each sentence and ignoring the grammar, we can see that sentences 1 and 3 are positive ("loved," "fun," "best") and sentences 2 and 4 are negative ("broke," "cheap," "flimsy").

Naive Bayes

Naive Bayes is a technique that's can be used in sentiment analysis with the bag-of-words model. In sentiment analysis, we are asking "What is the probability that the sentence is positive/negative given the words in the sentence." Answering this question requires computing conditional probability, and it is helpful to recall Bayes' rule:

$$P(B \vee A) = \frac{P(B)P(A \vee B)}{P(A)}$$

Now, we would like to use this formula to find $P(\text{sentiment} \mid \text{text})$, or, for example, $P(\text{positive} \mid \text{"my grandson loved it"})$. We start by tokenizing the input, such that we end up with $P(\text{positive} \mid \text{"my", "grandson", "loved", "it"})$. Applying Bayes' ruled directly, we get the following expression: $P(\text{"my", "grandson", "loved", "it"} \mid \text{positive}) * P(\text{positive}) / P(\text{"my", "grandson", "loved", "it"})$. This complicated expression will give us the precise answer to $P(\text{positive} \mid \text{"my", "grandson", "loved", "it"})$.

However, we can simplify the expression if we are willing to get an answer that's not equal, but proportional to $P(\text{positive} \mid \text{"my", "grandson", "loved", "it"})$. Later on, knowing that the probability distribution needs to sum up to 1, we can normalize the resulting value into an exact probability. This means that we can simplify the expression above to the numerator only: $P(\text{"my", "grandson", "loved", "it"} \mid \text{positive}) * P(\text{positive})$. Again, we can simplify this expression based on the knowledge that a conditional probability of a given b is proportional to the joint probability of a and b. Thus, we get the following expression for our probability: $P(\text{positive, "my", "grandson", "loved", "it"}) * P(\text{positive})$. Calculating this joint probability, however, is complicated, because the probability of each word is conditioned on the probabilities of the words preceding it. It requires us to compute $P(\text{positive}) * P(\text{"my"} \mid \text{positive}) * P(\text{"grandson"} \mid \text{positive, "my"}) * P(\text{loved} \mid \text{positive, "my", "grandson"}) * P(\text{"it"} \mid \text{positive, "my", "grandson", "loved"})$.

Here is where we use Bayes' rules naively: we assume that the probability of each word is independent from other words. This is not true, but despite this imprecision, Naive Bayes' produces a good sentiment estimate. Using this assumption, we end up with the following probability: $P(\text{positive}) * P(\text{"my"} \mid \text{positive}) * P(\text{"grandson"} \mid \text{positive}) * P(\text{"loved"} \mid \text{positive}) * P(\text{"it"} \mid \text{positive})$, which is not that difficult to calculate. $P(\text{positive})$ = the number of all positive samples divided by the number of total samples. $P(\text{"loved"} \mid \text{positive})$ is equal to the number of positive samples with the word "loved" divided by the number of positive samples. Let's consider the example below, with smiling and frowning emojis substituting the words "positive" and "negative":

				
0.49	0.51			
 0.00014112		my	0.30	0.20
 0.00006528		grandson	0.01	0.02
		loved	0.32	0.08
		it	0.30	0.40

On the right we are seeing a table with the conditional probabilities of each word on the left occurring in a sentence given that the sentence is positive or negative. In the small table on the left we are seeing the probability of a positive or a negative sentence. On the bottom left we are seeing the resulting probabilities following the computation. At this point, they are in proportion to each other, but they don't tell us much in terms of probabilities. To get the probabilities, we need to normalize the values, arriving at $P(\text{positive}) = 0.6837$ and $P(\text{negative}) = 0.3163$. The strength of naive Bayes is that it is sensitive to words that occur more often in one type of sentence than in the other. In our case, the word "loved" occurs much more often in positive sentences, which makes the whole sentence more likely to be positive than negative. To see an implementation of sentiment assessment using Naive Bayes with the nltk library, refer to `sentiment.py`.

One problem that we can run into is that some words may never appear in a certain type of sentence. Suppose none of the positive sentences in our sample had the word "grandson." Then, $P(\text{"grandson"} \mid \text{positive}) = 0$, and when computing the probability of the sentence being positive we will get 0. However, this is not the case in reality (not all sentences mentioning grandsons are negative). One way to go about this problem is with *Additive Smoothing*, where we add a value α to each value in our distribution to smooth the data. This way, even if a certain value is 0, by adding α to it we won't be multiplying the whole probability for a positive or negative sentence by 0. A specific type of additive smoothing, *Laplace Smoothing* adds 1 to each value in our distribution, pretending that all values have been observed at least once.

##4. Word Representation

We want to represent word meanings in our AI. As we've seen before, it is convenient to provide input to the AI in the form of numbers. One way to go about this is by using **One-Hot Representation**, where each word is represented with a vector that consists of as many values as we have words. Except for a single value in the vector that is equal to 1, all other values are equal to 0. How we can differentiate words is by which of the values is 1, ending up with a unique vector per word. For example, the sentence "He wrote a book" can be represented as four vectors:

```
[1, 0, 0, 0] (he)
[0, 1, 0, 0] (wrote)
[0, 0, 1, 0] (a)
[0, 0, 0, 1] (book)
```

However, while this representation works in a world with four words, if we want to represent words from a dictionary, when we can have 50,000 words, we will end up with 50,000 vectors of length 50,000. This is incredibly inefficient. Another problem in this kind of

representation is that we are unable to represent similarity between words like “wrote” and “authored.” Instead, we turn to the idea of **Distributed Representation**, where meaning is distributed across multiple values in a vector. With distributed representation, each vector has a limited number of values (much less than 50,000), taking the following form:

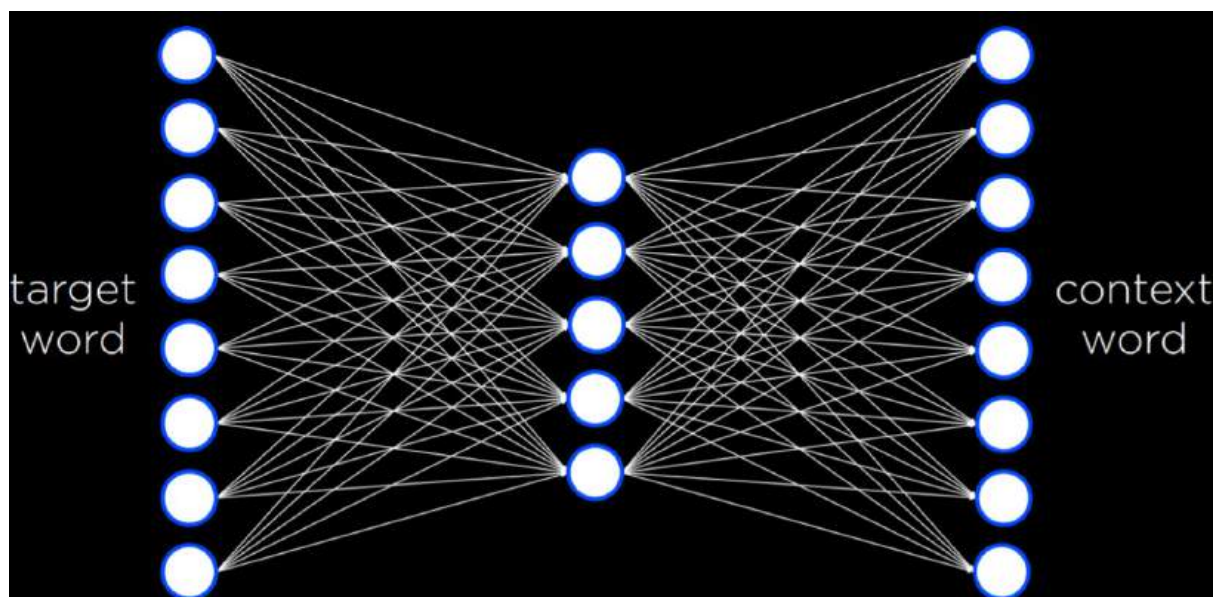
```
[ -0.34, -0.08, 0.02, -0.18, ...] (he)
[ -0.27, 0.40, 0.00, -0.65, ...] (wrote)
[ -0.12, -0.25, 0.29, -0.09, ...] (a)
[ -0.23, -0.16, -0.05, -0.57, ...] (book)
```

This allows us to generate unique values for each word while using smaller vectors. Additionally, now we are able to represent similarity between words by how different the values in their vectors are.

“You shall know a word by the company it keeps” is an idea by J. R. Firth, an English linguist. Following this idea, we can come to define words by their adjacent words. For example, there are limited words that we can use to complete the sentence “for ___ he ate.” These words are probably words like “breakfast,” “lunch,” and “dinner.” This brings us to the conclusion that by considering the environment in which a certain word tends to appear, we can infer the meaning of the word.

word2vec

word2vec is an algorithm for generating distributed representations of words. It does so by Skip-Gram Architecture, which is a neural network architecture for predicting context given a target word. In this architecture, the neural network has an input unit for every target word. A smaller, single hidden layer (e.g. 50 or 100 units, though this number is flexible) will generate values that represent the distributed representations of words. Every unit in this hidden layer is connected to every unit in the input layer. The output layer will generate words that are likely to appear in a similar context as the target words. Similar to what we saw in last lecture, this network needs to be trained with a training dataset using the backpropagation algorithm.



This neural network turns out to be quite powerful. In the end, of the process, every word ends up being just a vector, or a sequence of numbers. For example,

```
book: [-0.226776 -0.155999 -0.048995 -0.569774 0.053220 0.124401 -
0.091108 -0.606255 -0.114630 0.473384 0.061061 0.551323 -0.245151 -
0.014248 -0.210003 0.316162 0.340426 0.232053 0.386477 -0.025104 -
0.024492 0.342590 0.205586 -0.554390 -0.037832 -0.212766 -0.048781 -
0.088652 0.042722 0.000270 0.356324 0.212374 -0.188433 0.196112 -
0.223294 -0.014591 0.067874 -0.448922 -0.290960 -0.036474 -0.148416
0.448422 0.016454 0.071613 -0.078306 0.035400 0.330418 0.293890
0.202701 0.555509 0.447660 -0.361554 -0.266283 -0.134947 0.105315
0.131263 0.548085 -0.195238 0.062958 -0.011117 -0.226676 0.050336 -
0.295650 -0.201271 0.014450 0.026845 0.403077 -0.221277 -0.236224
0.213415 -0.163396 -0.218948 -0.242459 -0.346984 0.282615 0.014165 -
0.342011 0.370489 -0.372362 0.102479 0.547047 0.020831 -0.202521 -
0.180814 0.035923 -0.296322 -0.062603 0.232734 0.191323 0.251916
0.150993 -0.024009 0.129037 -0.033097 0.029713 0.125488 -0.018356 -
0.226277 0.437586 0.004913]
```

By themselves, these numbers don't mean much. But by finding which other words in the corpus have the most similar vectors, we can run a function that will generate the words that are the most similar to the word book. In the case of this network it will be: book, books, essay, memoir, essays, novella, anthology, blurb, autobiography, audiobook. This is not bad for a computer! Through a bunch of numbers that don't carry any specific meaning themselves, the AI is able to generate words that really are very similar to book not in letters or sounds, but in meaning! We can also compute the difference between words based on how different their vectors are. For example, the difference between king and man is similar to the difference between queen and woman. That is, if we add the difference between king and man to the vector for woman, the closest word to the resulting vector is queen! Similarly, if we add the difference between ramen and japan to america, we get burritos. By using neural networks and distributed representations for words, we get our AI to understand semantic similarities between words in the language, bringing us one step closer to AIs that can understand and produce human language.

##5. Practical NLP Applications

Neural Networks Recall that a neural network takes some input, passes it to the network, and creates some output. By providing the network with training data, it can do more and more of an accurate job of translating the input into an output. Commonly, machine translation uses neural networks. In practice, when we are translating words, we want to translate a sentence or paragraph. Since a sentence is a fixed size, we run into the problem of translating a sequence to another sequence where sizes are not fixed. If you have ever had a conversation with an AI chatbot, it needs to understand a sequence of words and generate an appropriate sequence as output.

Recurrent neural networks can re-run the neural network multiple times, keeping track of a state that holds all relevant information. Input is taken into the network, creating a hidden state. Passing a second input into the encoder, along with the first hidden state, produces a new hidden state. This process is repeated until an end token is passed. Then, a decoding state begins, creating hidden state after hidden state until we get the final word and another end token. Some problems, however, arise. One problem in the encoder stage where all the information from the input stage must be stored in one final state. For large sequences, it's very challenging to store all that information into a single state value. It would be useful to somehow combine all the hidden states. Another problem is that some of the hidden states in the input sequence are more important than others. Could there be some way to know what states (or words) are more important than others?

Attention Attention refers to the neural network's ability to decide what values are more important than others. In the sentence "What is the capital of Massachusetts," attention allows the neural network to decide what values it will pay attention to at each stage of generating the output sentence. Running such a calculation, the neural network will show that when generating the final word of the answer, "capital" and "Massachusetts" are the most important to pay attention to. By taking the attention score, multiplying them by the hidden state values generated by the network, and adding them up, the neural network will create a final context vector that the decoder can use to calculate the final word. A challenge that arises in calculations such as these is that recurrent neural networks require sequential training of word after word. This takes a lot of time. With the growth of large language models, they take longer and longer to train. A desire for parallelism has steadily grown as larger and larger datasets need to be trained. Hence, a new architecture has been introduced.

Transformers Transformers is a new type of training architecture whereby each input word is passed through a neural network simultaneously. An input word goes into the neural network and is captured as an encoded representation. Because all words are fed into the neural network at the same time, word order could easily be lost. Accordingly, position encoding is added to the inputs. The neural network, therefore, will use both the word and the position of the word in the encoded representation. Additionally, a self-attention step is added to help define the context of the word being inputted. In fact, neural networks will often use multiple self-attention steps such that they can further understand the context. This process is repeated multiple times for each of the words in the sequence. What results are encoded representations that will be useful when it's time to decode the information.

In the decoding step, the previous output word and its positional encoding are given to multiple self-attention steps and the neural network. Additionally, multiple attention steps are fed the encoded representation from the encoding process and provided to the neural network. Hence, words are able to pay attention to each other. Further, parallel processing is possible, and the calculations are fast and accurate.

6. Summing Up

Natural Language Processing (NLP), a dynamic subfield of artificial intelligence (AI), has ushered in a revolution in human-computer interaction by enabling machines to understand, interpret, and generate human language. In this exploration of NLP in AI, we've journeyed through its key components, applications, challenges, and future prospects, revealing the profound impact it has had on our daily lives.

At its core, NLP is the bridge that connects human language with machines. It encompasses a range of techniques and algorithms that empower computers to comprehend and process natural language, be it in the form of text or speech. From chatbots that provide customer support to language translation services and voice assistants, NLP is reshaping the way we communicate with technology.

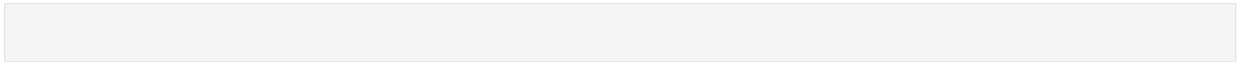
The applications of NLP are diverse and far-reaching. In the realm of communication, it enables instant language translation, breaking down linguistic barriers and fostering global connectivity. In healthcare, NLP aids in clinical documentation, medical diagnosis, and data analysis, improving patient care. Sentiment analysis and social media monitoring harness NLP to gauge public opinion and trends, benefiting businesses and policymakers alike. Content recommendation systems, powered by NLP, enhance user experiences on digital platforms, from streaming services to e-commerce websites.

However, NLP is not without its challenges. Ambiguity, context, and cultural nuances pose hurdles in accurately interpreting human language. Privacy and ethical concerns arise in the

context of data collection and utilization. Bias and fairness issues demand constant vigilance and ethical scrutiny to ensure NLP systems do not perpetuate discriminatory practices.

As we look to the future of NLP in AI, there is immense promise. Deep learning models, such as transformers, have propelled NLP to new heights, enabling more nuanced understanding of context and semantics. Multilingual models promise to further break down language barriers, while low-resource language support will democratize NLP applications globally. Advancements in neural speech synthesis will continue to make voice assistants more natural and intuitive.

In conclusion, NLP in AI is reshaping the way we interact with technology and each other. Its transformative power extends across industries, from healthcare to education, entertainment, and beyond. As researchers and practitioners continue to push the boundaries of what NLP can achieve, we can anticipate a future where human-computer communication is seamless, language is no longer a barrier, and the benefits of NLP enrich our lives in ways we can only imagine today.



Natural Language Processing (NLP). Practical Tasks

You task is to implement all the steps of NLP algorithm

```
!pip install git+https://github.com/mehalyna/cooltest.git
from cooltest.test_cool_1 import *

#install all the needed modules
#import spacy library

#load core english library
```

1. Segmentation.

Break the entire document down into its constituent sentences. You can do this by segmenting the article along with its punctuations like full stops and commas.

```
text = '''Natural Language Processing (NLP) is a part of AI
(artificial intelligence) that deals with understanding and
processing of human language.
In real time, majority of data exists in the unstructured form such
us text, videos, images.
Mass of data in unstructured category, will be in textual form.
To process this textual data's with machine learning algorithms, NLP
comes in to play.
NLP use cases are Language translation, Speech recognition, Hiring
and Recruitment, Chat Bot, Sentimental analysis and so on.'''
```

Write the body of function `get_segment` which would be used for the segmentation of the given `text`. Function will return the sequence of the segments as the result.

```
@test_get_segment
def get_segment(some_text):
    #take string

    #to print sentences

    return sentence
```

1. Tokenizing

Define the body of the function `get_tokens()` that would be used for breaking down given sentence into its constituent words (tokens). The function will return the sequence of the tokens as the result.

```
# import Tokenizer
```

```
text = "Natural Language Processing (NLP) is a part of AI  
(artificial intelligence) that deals with understanding and  
processing of human language."
```

```
@test_get_tokens  
def get_tokens(some_text):  
  
    return tokens
```

1. Removing Stop Words

Your next task is getting rid of non-essential words, which add little meaning to the statement `new_tokens` and are just there to make your statement sound more cohesive. Words such as *was, in, is, and, the*, are called stop words and can be removed. Write the body of function `remove_stop_words()` which will take the parameter sequence with tokens (`some_tokens`) and return as the result the sequence of tokens without non-essential words (*stop words*)

```
# import module for stop words removing  
  
@test_remove_stop_words  
def remove_stop_words(some_tokens):  
  
    return filtered_sentence
```

1. Stemming and Lemmatization

Now you have to obtain the Root Stem of a word. Root Stem gives the new base form of a word that is present in the dictionary and from which the word is derived. You can also identify the base word for different words, - lemma.

Write the body `stemm_lemmatization()` of function which should return the list of lemmas of given parameter text.

```
@test_stemm_lemmatization  
def stemm_lemmatization(doc):  
  
    return list_lemma
```

1. Part of Speech Tagging:

Now, you have to define the body of function `part_of()` that will take the sentence of text as the parameter. The function should return the dictionary with words as keys and appropriate tags (parts of speech) as values.

```
@test_part_of  
def part_of(doc):  
  
    return word_tags
```

Computer Vision in AI

Duration: 2 hours

1. Introduction
2. Fundamentals of Computer Vision
3. Image Processing Techniques
4. Object Detection
5. Image Classification

1. Introduction

The phenomenon that makes machines such as computers or mobile phones see the surroundings is known as **Computer Vision**. Serious work on re-creating a human eye started way back in 50s and since then, we have come a long way. Computer vision has already made its way to our mobile phone via different e-commerce or camera apps.

Computer vision is a fascinating field that empowers computers to see and understand the visual world just like humans do. It involves developing algorithms and techniques to extract meaningful information from images or videos. By leveraging the power of artificial intelligence, computer vision enables machines to perceive, analyze, and interpret visual data, opening up a wide range of possibilities for various applications.

In the realm of artificial intelligence, computer vision plays a crucial role. It allows machines to process and make sense of the vast amount of visual information available, making it an integral part of many AI systems. With computer vision, AI algorithms can recognize and classify objects, understand scenes, track movements, and even comprehend facial expressions. This capability enables machines to interact with the world in a more intelligent and human-like manner.

Computer vision finds applications in numerous domains. In the field of robotics, computer vision enables robots to navigate and manipulate objects in their environment. In autonomous vehicles, it assists in recognizing traffic signs, pedestrians, and obstacles, ensuring safe and efficient navigation. In healthcare, computer vision aids in medical imaging analysis, assisting doctors in diagnosing diseases and conditions. It also enhances security systems by enabling facial recognition and object detection for surveillance purposes.

By harnessing computer vision techniques, we can unlock tremendous potential in various industries, from retail and manufacturing to entertainment and healthcare. It revolutionizes the way we perceive and interact with the visual world. Understanding computer vision is essential for anyone interested in artificial intelligence as it forms the basis for many advanced AI applications.

[Slides](#) for better understanding.

Think of what more can be done by machine when they will be able to see as accurate as a human eye. Human eye is a complex structure and it goes through more complex phenomenon of understanding the environment. In a similar fashion, making machines see things and make them capable enough to figure out what they are seeing and further categorize it, is still a pretty tough job.

Working on Computer Vision is equivalent to working on millions of calculations in the blink of an eye with almost same accuracy as that of a human eye. It is not just about converting a picture into pixels, and then try to make sense of what's in the picture through those pixels, you will have to first understand the bigger picture of how to extract information from those pixels and understand what they represent.

Throughout this session, we will explore the fundamental concepts, techniques, and applications of computer vision, providing you with a solid foundation to embark on your journey into this exciting field.

Let's look at one example of Computer Vision Algorithm.

IMPORTANT: Run the scripts in this section first and run them in order; otherwise, code in subsequent sections may fail or hang.

```
!pip install opencv-contrib-python --upgrade

# %load ../compat/jupyter_compat.py
import os

import cv2
import numpy
import PIL.Image

from IPython import import display
from urllib.request import import urlopen

def cv2_imshow(winname, mat):
    mat = mat.clip(0, 255).astype('uint8')
    if mat.ndim == 3:
        if mat.shape[2] == 4:
            mat = cv2.cvtColor(mat, cv2.COLOR_BGRA2RGBA)
        else:
            mat = cv2.cvtColor(mat, cv2.COLOR_BGR2RGB)
    display.display(PIL.Image.fromarray(mat))

cv2.imshow = cv2_imshow

def cv2_waitKey(delay=0):
    return -1

cv2.waitKey = cv2_waitKey

def cv2_imread(filename, flags=cv2.IMREAD_COLOR):
    if os.path.exists(filename):
        image = cv2.imread(filename, flags)
    else:
        url =
f'https://github.com/mehalyna/CV_for_Beginners/raw/main/{filename}'
        resp = urlopen(url)
        image = numpy.asarray(bytearray(resp.read()), dtype='uint8')
        image = cv2.imdecode(image, flags)
```

```

    return image

if '_imread' not in dir(cv2):
    cv2._imread = cv2.imread

cv2.imread = cv2_imread

# %load face_detection_still.py
#import cv2

face_cascade = cv2.CascadeClassifier(
    f'{cv2.data.harcascades}haarcascade_frontalface_default.xml')
img = cv2.imread('../images/woodcutters.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
faces = face_cascade.detectMultiScale(gray, 1.08, 5)
for (x, y, w, h) in faces:
    img = cv2.rectangle(img, (x, y), (x+w, y+h), (255, 255, 0), 2)

cv2.imshow('Woodcutters Detected!', img)
cv2.imwrite('./woodcutters_detected.png', img)
cv2.waitKey(0)

```



- 1

2. Fundamentals of Computer Vision

Computer vision is an interdisciplinary field of study that focuses on enabling computers to interpret and understand visual information from the world around us. It involves developing algorithms, techniques, and models that allow machines to process, analyze, and make sense of images or videos, much like the human visual system.

At its core, computer vision seeks to bridge the gap between the raw pixel data in images and the meaningful information that humans can extract from them. By mimicking the human visual perception process, computer vision algorithms can identify objects, detect patterns, recognize faces, understand scenes, and even estimate 3D structures from 2D images.

Applications of Computer Vision:

1. *Image Recognition and Classification:* Computer vision is extensively used for image recognition tasks. It can categorize images into predefined classes or labels, such as identifying different objects, animals, or scenes. This application

has numerous practical uses, including content-based image retrieval, quality control in manufacturing, and automatic image tagging in social media.

2. *Object Detection:* Object detection is a crucial task in computer vision, where algorithms identify and locate specific objects within an image or video stream. This capability is essential in applications like autonomous vehicles, surveillance systems, and robotics, enabling machines to understand and interact with their environment effectively.
3. *Facial Recognition:* Computer vision plays a pivotal role in facial recognition systems, allowing machines to identify and verify individuals based on their unique facial features. This application finds use in security systems, user authentication, and even in personalized customer experiences, such as unlocking smartphones or tagging people in photos.
4. *Medical Imaging and Healthcare:* In the medical field, computer vision assists in analyzing medical images, including X-rays, MRIs, and CT scans. It helps detect anomalies, tumors, and other medical conditions, aiding doctors in diagnosis and treatment planning.
5. *Augmented Reality (AR) and Virtual Reality (VR):* Computer vision is fundamental to AR and VR technologies, enabling virtual objects to be superimposed or integrated into the real-world environment. This allows for interactive and immersive experiences in gaming, training simulations, and architectural visualization.
6. *Autonomous Vehicles:* The development of self-driving cars heavily relies on computer vision techniques. By processing data from cameras, lidar, and other sensors, autonomous vehicles can navigate, detect obstacles, and make decisions based on their surroundings.
7. *Gesture Recognition:* Computer vision enables machines to interpret human gestures, such as hand movements or facial expressions, enabling natural and intuitive interactions in various applications, including gaming, sign language recognition, and human-robot interactions.

As technology continues to advance, the scope of computer vision applications will only expand further, transforming industries and enhancing various aspects of our daily lives. The possibilities are vast, making computer vision a compelling and essential field within the broader domain of artificial intelligence.

Key Concepts of Computer Vision

1. **Image Processing:** Image processing is the fundamental concept in computer vision that involves manipulating and enhancing images to extract useful information or improve their quality for subsequent analysis. It encompasses a wide range of techniques and algorithms that operate on images, typically in a pixel-by-pixel manner. The primary goals of image processing are noise reduction, image enhancement, and feature extraction. *Key Techniques in Image Processing:*
 - *Image Filtering:* Filtering operations are used to smooth or sharpen an image by applying various kernels or filters to modify the pixel values. Common filters include Gaussian blur for smoothing and edge detection filters for finding edges in the image.

- *Image Transformation*: Image transformations involve converting an image into a different representation. For instance, converting a color image to grayscale or applying mathematical operations like rotation, scaling, and translation.
 - *Image Restoration*: Image restoration techniques aim to remove noise or artifacts from images, making them clearer and easier to analyze.
 - *Image Segmentation*: Image segmentation divides an image into meaningful regions or segments, which aids in isolating objects or regions of interest for further analysis.
2. **Object Detection**: Object detection is a computer vision task that involves locating and identifying objects of interest within an image or a video frame. Unlike image classification, where the goal is to assign a single label to the entire image, object detection aims to draw bounding boxes around multiple objects and classify each object within those boxes. This capability is crucial for various applications, including autonomous vehicles, surveillance, and robotics. *Key Techniques in Object Detection*:
- *Sliding Window Technique*: Involves scanning the image with a window of fixed size and aspect ratio and classifying each window as containing an object or not.
 - *Feature-Based Methods*: These methods use handcrafted features, like Histogram of Oriented Gradients (HOG) or Haar-like features, to represent objects and enable detection.
 - *Deep Learning-based Approaches*: Convolutional Neural Networks (CNNs) have revolutionized object detection. Models like R-CNN, Fast R-CNN, and YOLO utilize CNNs to detect objects accurately and efficiently.
3. **Image Classification**: Image classification is the process of assigning a label or category to an entire image based on its content. Given a set of predefined classes, the algorithm determines the most likely class that the image belongs to. Image classification is one of the foundational tasks in computer vision and has applications in diverse fields, including medical diagnosis, natural language processing, and autonomous vehicles. *Key Techniques in Image Classification*:
- *Feature Extraction*: Initially, handcrafted features like SIFT or SURF were used for image representation. However, deep learning-based approaches using CNNs have become the state-of-the-art for feature extraction in image classification tasks.
 - *Convolutional Neural Networks (CNNs)*: CNNs have shown exceptional performance in image classification. They automatically learn hierarchical features from raw pixel data, enabling highly accurate image recognition.
 - *Transfer Learning*: Transfer learning is a technique where pre-trained CNN models on large datasets like ImageNet are used as a starting point for new image classification tasks with smaller datasets. This approach allows for better performance with limited data.

These key concepts of image processing, object detection, and image classification are the building blocks of many advanced computer vision applications. Understanding and mastering these concepts are essential for anyone working in the field of artificial intelligence and computer vision.

Common computer vision techniques include:

1. *Image Filtering* involves applying various filters or convolution kernels to an image to enhance specific features, reduce noise, or perform edge detection. Common filters include Gaussian blur, Sobel, and Laplacian.

2. *Image Segmentation* divides an image into distinct regions or segments based on similarities in color, texture, or intensity. This technique is useful for object isolation and boundary detection.
3. *Object Detection* algorithms locate and identify multiple objects within an image or video stream. Techniques like Haar cascades, Histogram of Oriented Gradients (HOG), and deep learning-based models (e.g., YOLO and SSD) are widely used.
4. *Image Classification* assigns a label or category to an entire image. Convolutional Neural Networks (CNNs) have become the cornerstone of image classification tasks due to their ability to learn hierarchical features.
5. *Feature Detection and Description* algorithms, such as Harris corners and FAST features, identify distinctive points in an image. Feature description methods like SIFT and SURF extract unique descriptors to match these points across images.
6. *Optical Character Recognition (OCR)* techniques enable computers to recognize and convert text within images into machine-readable text. Tesseract is a popular open-source OCR engine.
7. *Face Detection and Recognition* identifies faces within images or videos, while face recognition verifies or identifies individuals. Deep learning models, like OpenCV's DNN face detector and FaceNet, are widely employed for accurate results.
8. *Image Registration* aligns two or more images to enable comparisons or create panoramas. It is commonly used in medical imaging, remote sensing, and augmented reality.
9. *Object Tracking* algorithms follow and track the movement of objects across consecutive frames in a video. Techniques like Kalman filters and correlation-based tracking are employed.
10. *3D Reconstruction* aims to build a three-dimensional model from multiple 2D images. Structure-from-Motion (SfM) and Multi-View Stereo (MVS) are typical methods used for 3D reconstruction.

These techniques form the foundation of computer vision applications and continue to evolve with advancements in deep learning and other AI technologies. By combining and customizing these methods, developers and researchers can create powerful computer vision systems for a wide range of real-world use cases.

3. Image Processing Techniques (30 minutes)

- a. Introduction to image preprocessing steps
- b. Discussion on image enhancement, filtering, and noise removal techniques
- c. Demonstration of popular image processing algorithms

###Step1: Load the Dependencies

This section loads some required libraries used in this notebook: **numpy**, **pandas**, **cv2**, **skimage**, **PIL**, **matplotlib**

- [Numpy](#) is an array manipulation library, used for linear algebra, Fourier transform, and random number capabilities.
- [Pandas](#) is a library for data manipulation and data analysis.
- [CV2](#) is a library for computer vision tasks.
- [Skimage](#) is a library which supports image processing applications on python.
- [Matplotlib](#) is a library which generates figures and provides graphical user interface toolkit.

Useful intro about [OpenCV](#)

```
import numpy as np
import pandas as pd
import cv2 as cv
from google.colab.patches import cv2_imshow # for image display
from skimage import io
from PIL import Image
import matplotlib.pyplot as plt
```

###Step2: Read Image from Urls

In this step we will read images from urls, and display them using openCV, please note the difference when reading image in RGB and BGR format. The default input color channels are in BGR format for openCV.

Image source example: Google Image search for an image. Left-click one of the returned images, then right-click on the full image, and then select *Copy Image Address*.

```
# Create a list to store the urls of the images
urls = ["https://images.unsplash.com/photo-1560114928-40f1f1eb26a0?ixlib=rb-4.0.3&ixid=MnwxMjA3fDB8MHxleHBsb3JlLWZlZWR8Mnx8fGVufDB8fHx8&auto=format&fit=crop&w=600&q=60pg",
        "https://images.unsplash.com/photo-1595433707802-6b2626ef1c91?ixlib=rb-4.0.3&ixid=MnwxMjA3fDB8MHxleHBsb3JlLWZlZWR8MXx8fGVufDB8fHx8&w=1000&q=80",
        "https://images.unsplash.com/photo-1574144611937-0df059b5ef3e?ixlib=rb-4.0.3&ixid=MnwxMjA3fDB8MHxleHBsb3JlLWZlZWR8Nnx8fGVufDB8fHx8&auto=format&fit=crop&w=600&q=60"]
# Read and display the image
# loop over the image URLs, you could store several image urls in the list

for url in urls:
    image = io.imread(url)
    image_2 = cv.cvtColor(image, cv.COLOR_BGR2RGB)
    final_frame = cv.hconcat((image, image_2))
    cv2_imshow(final_frame)
    print('\n')
```





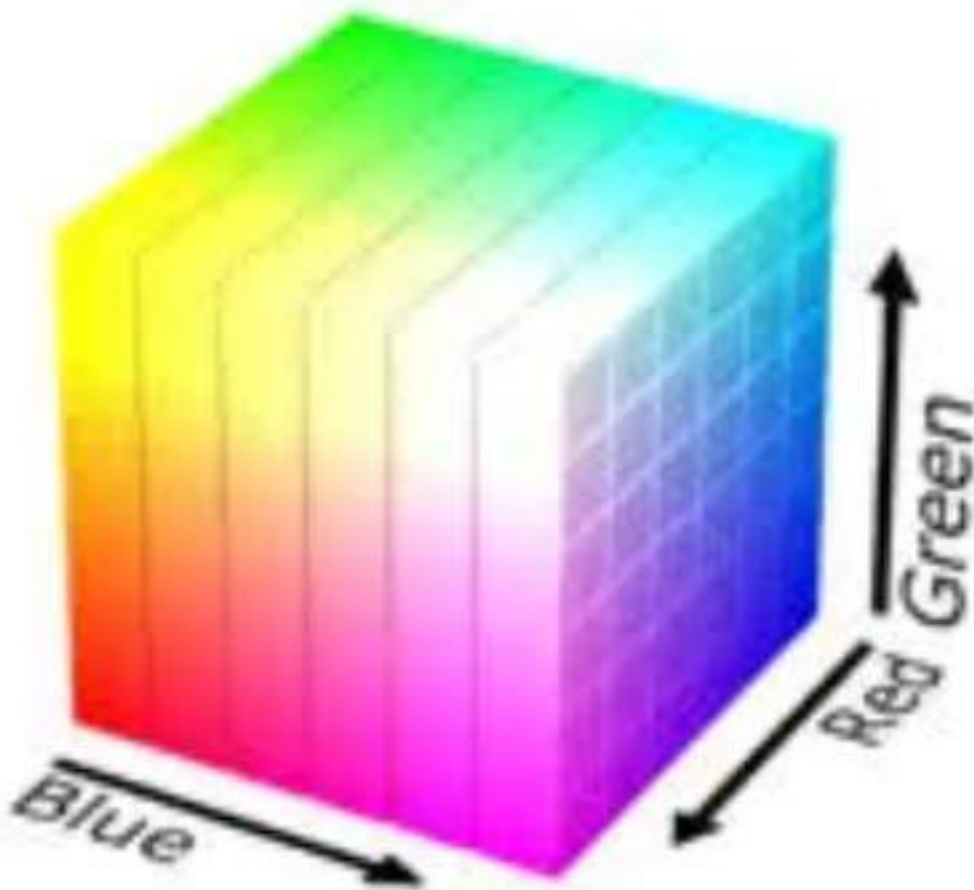
Step 3. Represent colors by numbers

In computer science, each color is represented by a specified HEX value. That is how machines are programmed to understand what colors the image pixels are made up. Whereas as humans we have an inherited knowledge to differ between the shades.

The four prominent color spaces provided by OpenCV are RGB, HSV, YCrCb and LAB. Each of the colour spaces have their properties and consideration of various factors affecting an image. Using the appropriate color space can help tackle problems involving color based segmentation.

RGB stands for the three colours - Red, Green and Blue. The pixels of an RGB image are made up with a linear combination of the colour intensities of Red, Green and Blue. The color differentiation is heavily dependent on the lighting conditions. Under bright lighting, the segregation of colours can be more visible as opposed to dull lighting. Hence, to perform colour based segmentation, RGB color space is not a good choice as the information related to color and luminance is mixed.

RGB color model



```
# Extracting blue channel and displaying
image = io.imread(urls[2])
image_2 = cv.cvtColor(image, cv.COLOR_BGR2RGB)
blue_image = image.copy()
blue_image[:, :, 0] = 0
cv2.imshow(blue_image)
#cv.waitKey(0)
```




```
# Extracting green channel and displaying  
image = io.imread(urls[2])  
image_2 = cv.cvtColor(image, cv.COLOR_BGR2RGB)  
green_image = image.copy()  
green_image[:, :, 1] = 0  
cv2_imshow(green_image)
```

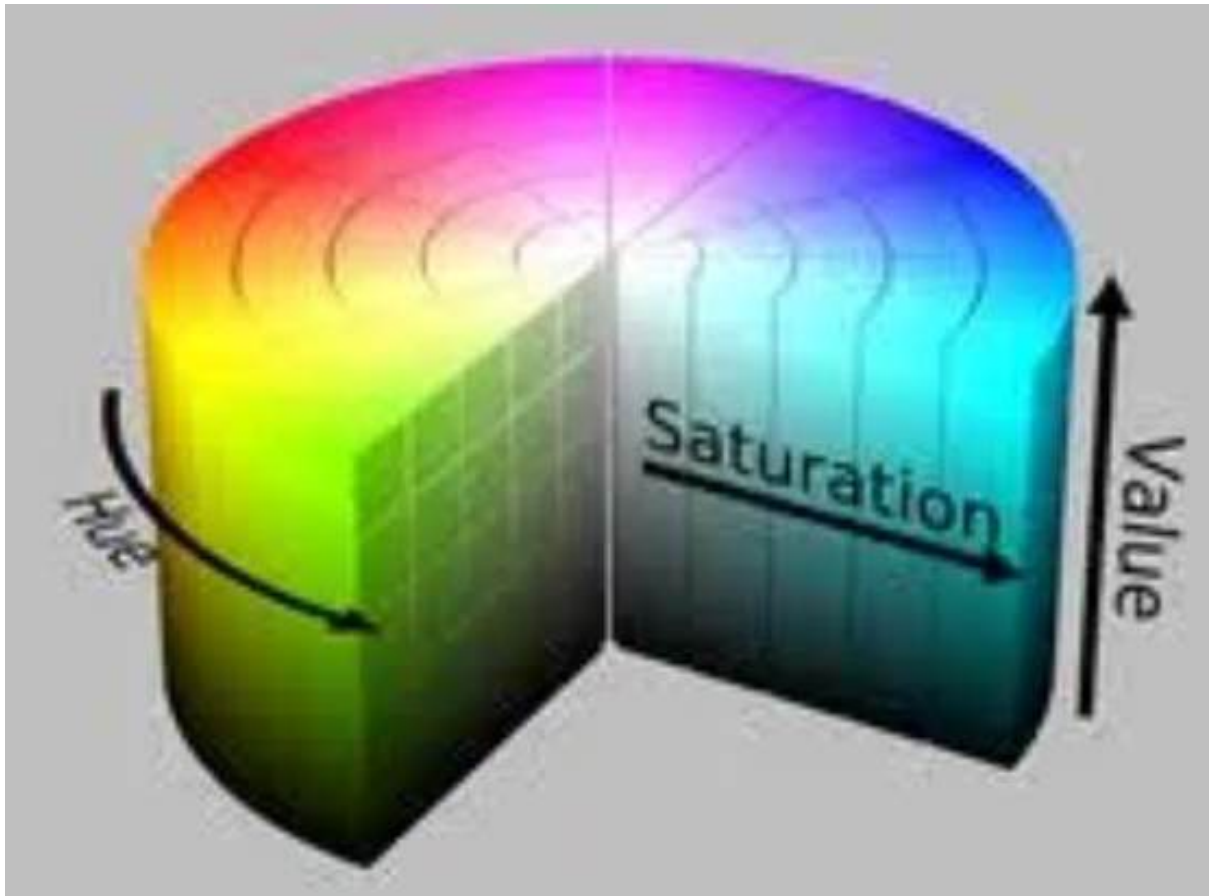


```
# Extracting red channel and displaying  
image = io.imread(urls[2])  
image_2 = cv.cvtColor(image, cv.COLOR_BGR2RGB)  
red_image = image.copy()  
red_image[:, :, 2] = 0  
cv2_imshow(red_image)
```

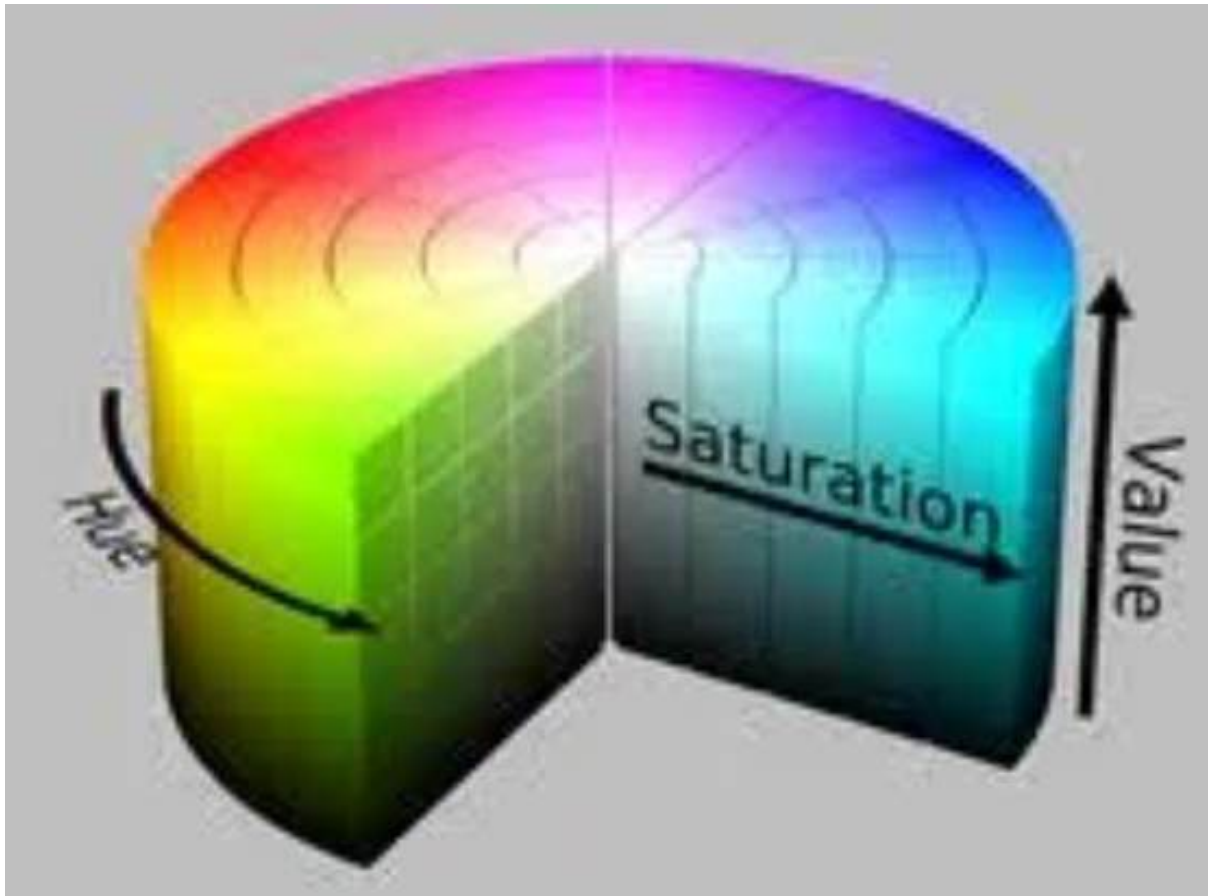


Later the HSV (Hue Saturation Value) and HSL (Hue Saturation Lightness) are developed and found to be more closely aligned to the way human perceives colors.

HSV color model



HSL color model



###Step3: Image Contours and Histograms

The algorithm for computer vision that involves defining image contours and histograms typically follows these steps:

1. Image Preprocessing:
 - Convert the input image to grayscale if it is in color.
 - Apply any necessary preprocessing techniques like noise removal, image enhancement, or smoothing to improve the quality of the image.
2. Edge Detection:
 - Apply edge detection algorithms such as Canny edge detection or Sobel operator to identify the edges in the image.
 - These algorithms highlight areas with significant intensity changes, which often correspond to object boundaries.
3. Contour Extraction:
 - Use contour detection algorithms like the Douglas-Peucker algorithm or the Marching Squares algorithm to extract contours from the detected edges.
 - Contours are represented as a sequence of points that form closed loops or outlines of objects in the image.
4. Contour Filtering and Analysis:
 - Apply filtering techniques to remove unwanted or noisy contours based on criteria such as contour area, length, or shape.
 - Analyze the remaining contours to extract useful information about objects in the image, such as their position, size, or orientation.

- Calculate additional properties of the contours, such as centroid, perimeter, or bounding box, to further characterize the objects.
5. Histogram Calculation:
 - Compute the histogram of the grayscale image or individual image channels.
 - The histogram represents the distribution of pixel intensities or color values in the image.
 - Depending on the application, you may choose to calculate a 1D histogram for grayscale images or separate histograms for each color channel in color images.
 6. Histogram Analysis:
 - Analyze the histogram to gain insights into the image's overall contrast, brightness, or color distribution.
 - Identify peaks or modes in the histogram that correspond to significant intensity or color ranges.
 - Use histogram equalization or other techniques to enhance the image's contrast or adjust its tonal distribution.

By following these steps, the algorithm can effectively identify image contours, which are important for object recognition, shape analysis, and other computer vision tasks. Additionally, calculating and analyzing histograms provides valuable information about the image's pixel intensity or color distribution, enabling further image processing and analysis.

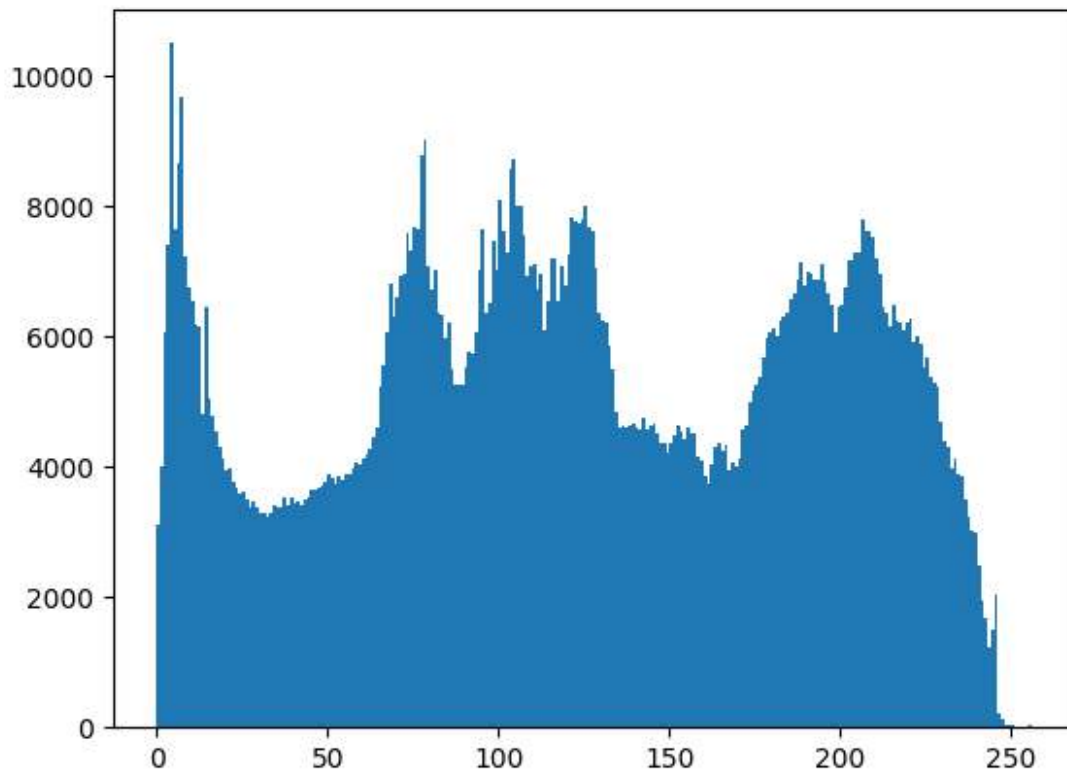
```
# Check the image matrix data type (could know the bit depth of the image)
print(image.dtype)
# Check the height of image
print(image.shape[0])
# Check the width of image
print(image.shape[1])
# Check the number of channels of the image
print(image.shape[2])

uint8
750
600
3
```

Generate Histogram of color image and grayscale image

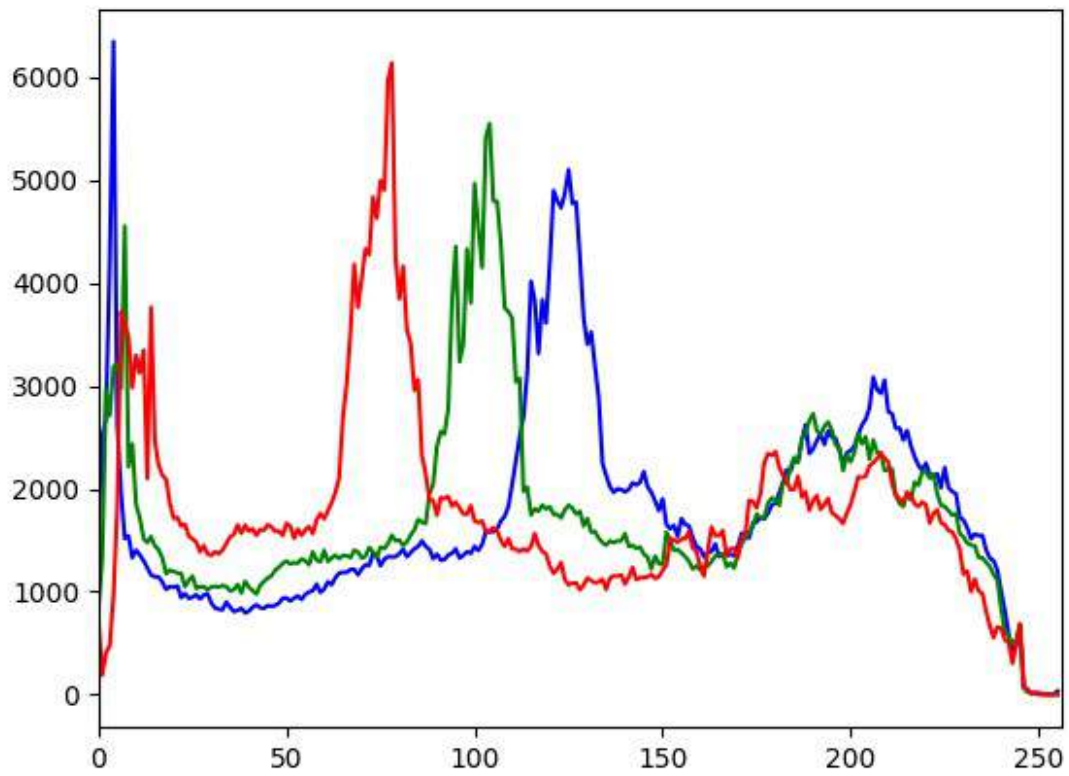
A good tool to find interesting regions is the histogram. To create a histogram of our image data, we use the `matplotlib.pyplot.hist()` function. Display the histogram of all the pixels in the color image:

```
plt.hist(image.ravel(), bins = 256, range = [0, 256])
plt.show()
```



Display the histogram of R, G, B channel We could observe that the green channel has many pixels in 255, which represents the white patch in the image

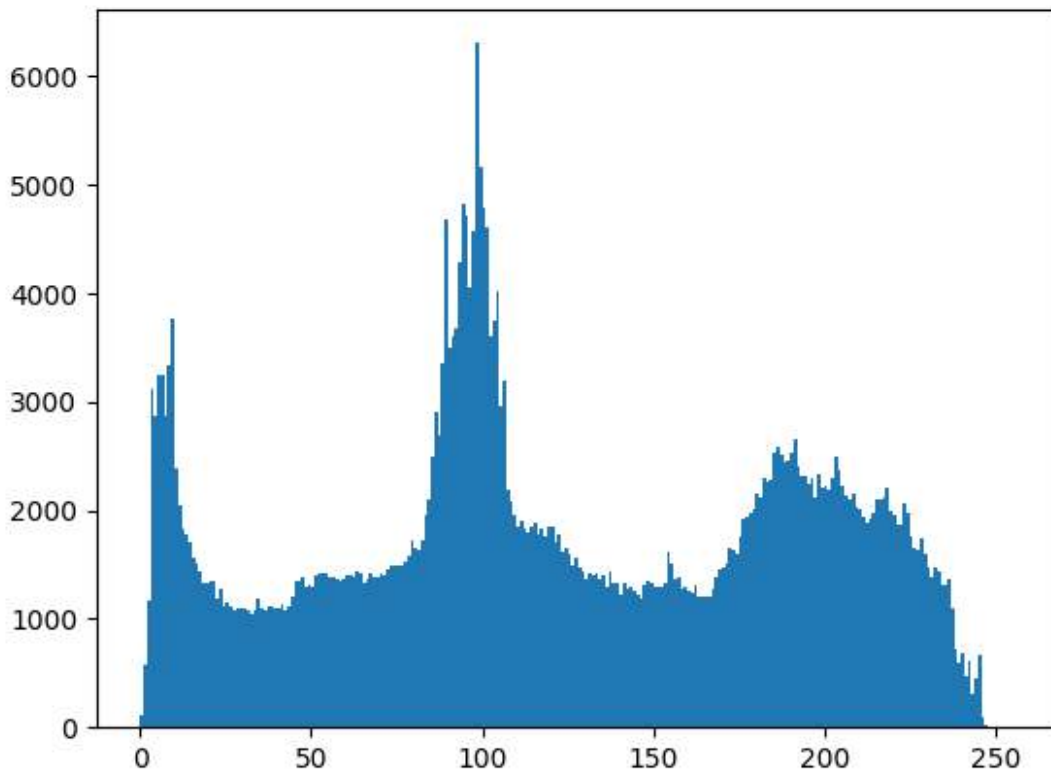
```
color = ('b', 'g', 'r')
for i, col in enumerate(color):
    histr = cv.calcHist([image], [i], None, [256], [0, 256])
    plt.plot(histr, color = col)
    plt.xlim([0, 256])
plt.show()
```



```
gray_image = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
cv2.imshow('gray_image', gray_image)
```



```
# Plot the histogram of the gray image. We could observe that the  
# frequency of  
# the image hist has decreased ~ 1/3 of the histogram of color image  
plt.hist(gray_image.ravel(),bins = 256, range = [0, 256])  
plt.show()
```

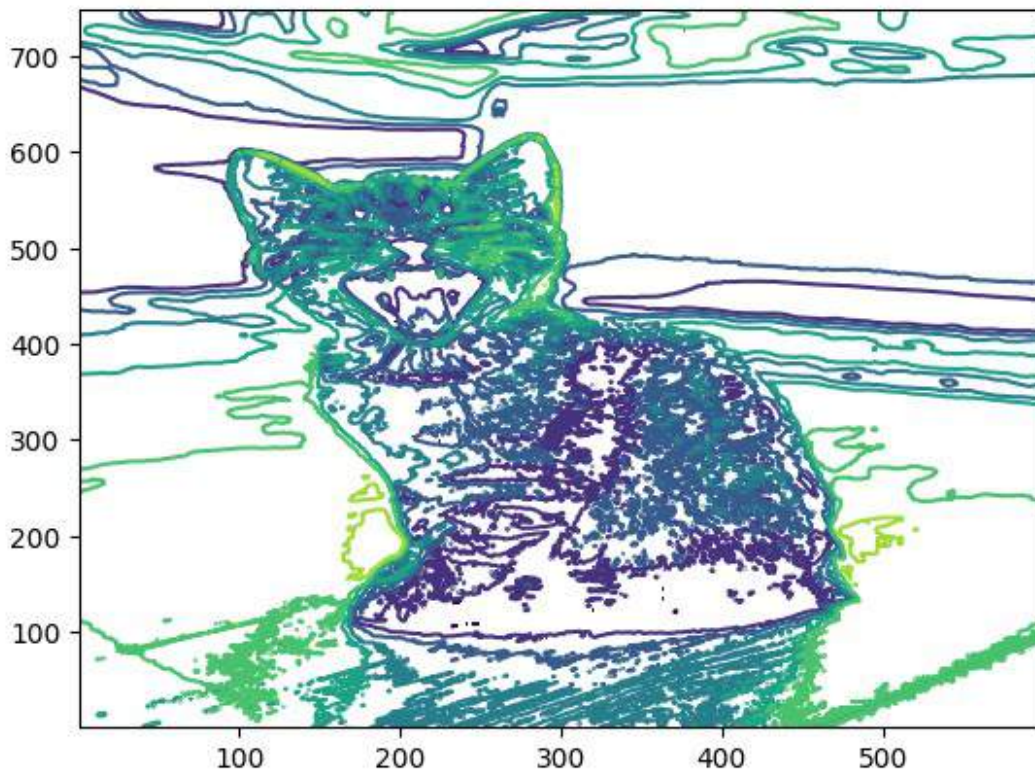


Find image contour of the grayscale image

Method 1: Use the matplotlib. contour

More Info: [matplotlib contour](#)

```
plt.contour(gray_image, origin = "image")  
<matplotlib.contour.QuadContourSet at 0x7867c9c7aa40>
```

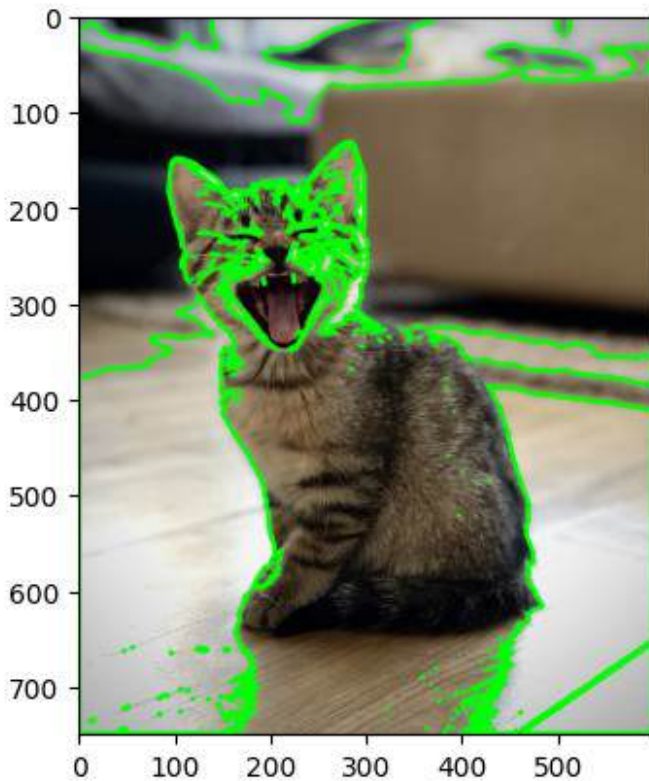



Method 2: Use the openCV lib

More info: [Contour](#)

```
# Set threshold for the countour detection
ret, thresh = cv.threshold(gray_image, 150, 255, 0)
contours, hierarchy = cv.findContours(thresh, cv.RETR_TREE,
cv.CHAIN_APPROX_SIMPLE)
cv.drawContours(image, contours, -1, (0, 255, 0), 3)
plt.imshow(image)
```

```
<matplotlib.image.AxesImage at 0x7867c9718cd0>
```

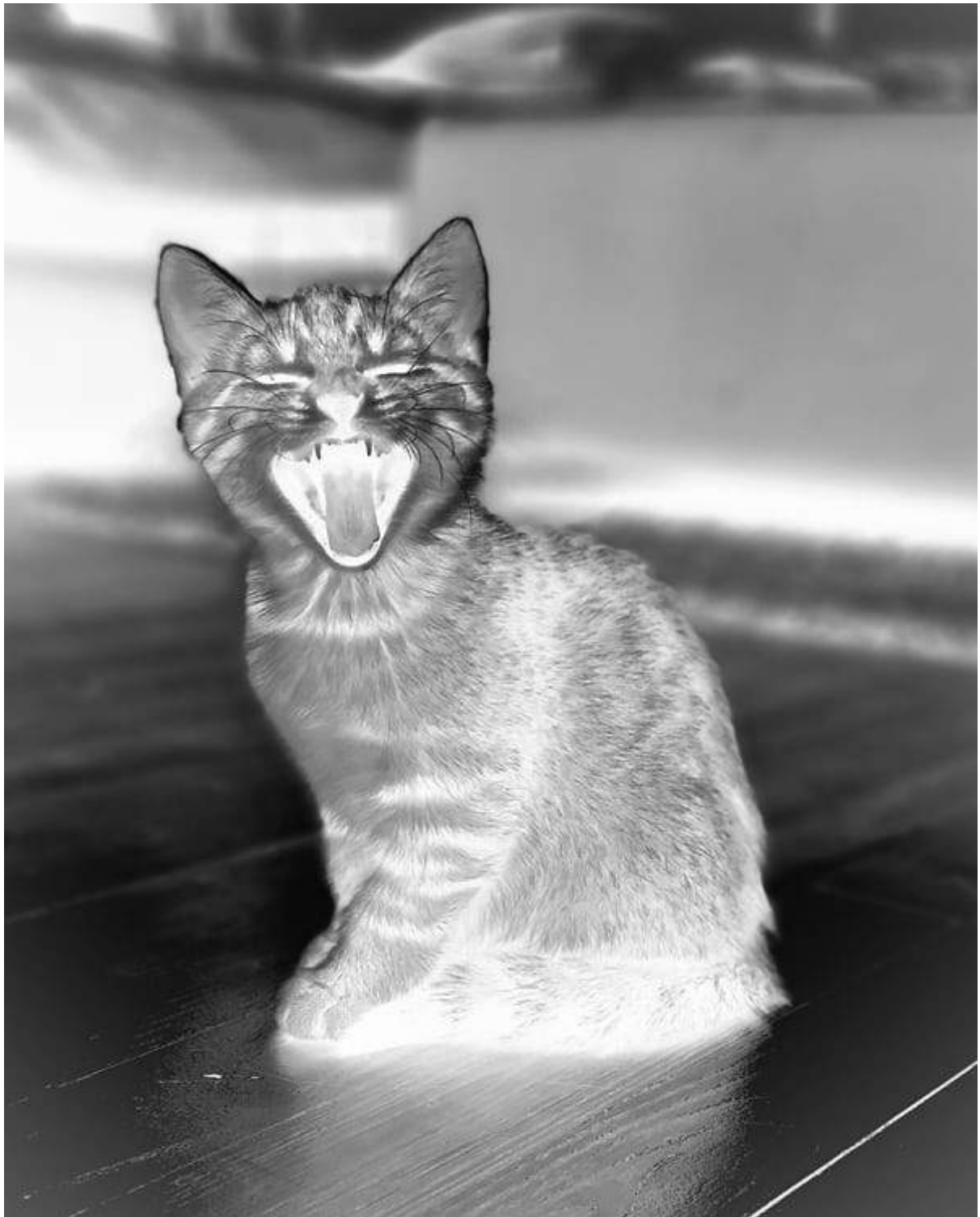



##Step4: Grayscale Transform and Histogram Equalization

Grayscale Transformation

This section provides some examples of conducting mathematical transformations of the grayscale image

```
# This is an inverse operation of the grayscale image, you could see that the  
# bright pixels become dark, and the dark pixels become bright  
im2 = 255 - gray_image  
cv2_imshow(im2)
```



```
# Another transform of the image, after adding a constant,  
# all the pixels become brighter and a hazing-like effect of the  
# image is generated  
im3 = (100.0/255)*gray_image + 100  
cv2_imshow(im3)
```



```
# The lightness level of the gray_image decreases after this step  
im4 = 255.0*(gray_image/255.0)**2  
cv2_imshow(im4)
```



Histogram Equalization

This section demonstrates histogram equalization on a dark image. This transform flattens the gray-level histogram so that all intensities are as equally common as possible. The transform function is a cumulative distribution function (cdf) of the pixel values in the image (normalized to map the range of pixel values to the desired range). This example uses image 4 (im4).

```

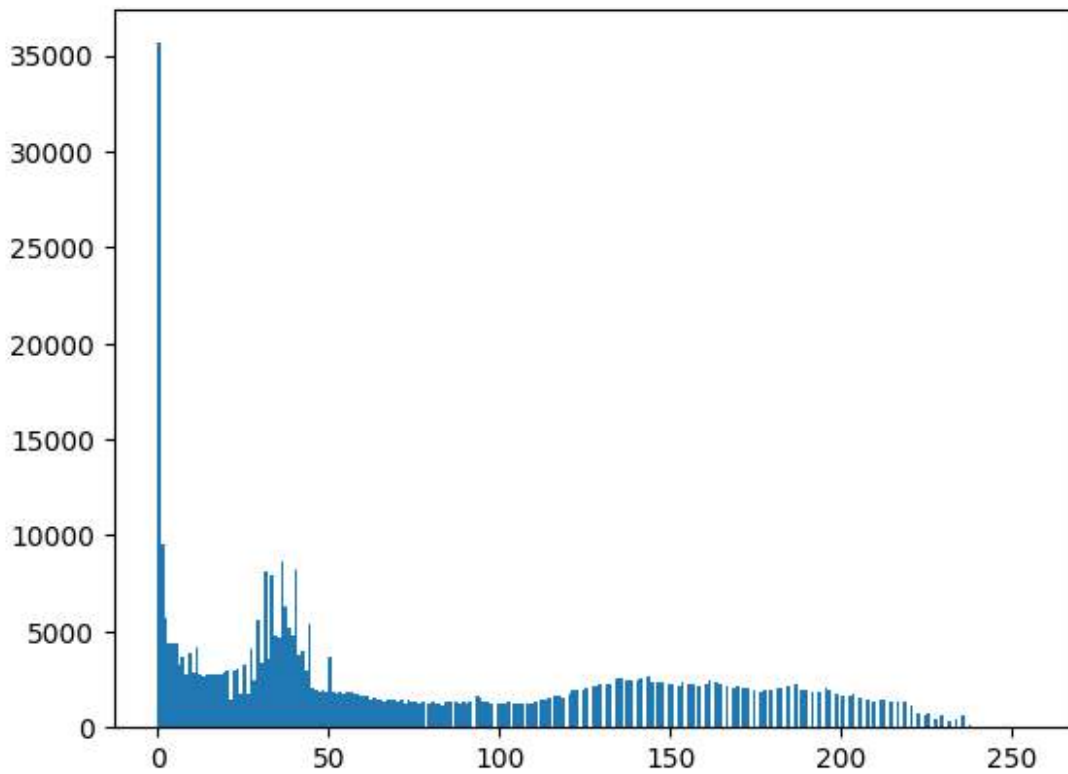
# function of histogram equalization
def histeq(im, nbr_bins = 256):
    """ Histogram equalization of a grayscale image. """
    # get the image histogram
    imhist, bins = np.histogram(im.flatten(), nbr_bins, [0, 256])
    cdf = imhist.cumsum() # cumulative distribution function
    cdf = imhist.max()*cdf/cdf.max() #normalize
    cdf_mask = np.ma.masked_equal(cdf, 0)
    cdf_mask = (cdf_mask - cdf_mask.min())*255/(cdf_mask.max() -
cdf_mask.min())
    cdf = np.ma.filled(cdf_mask,0).astype('uint8')
    return cdf[im.astype('uint8')]

# apply the function on your dark image to increase the contrast
# we could observe that the contrast of the black background has
increased
im5 = histeq(im4)
cv2_imshow(im5)

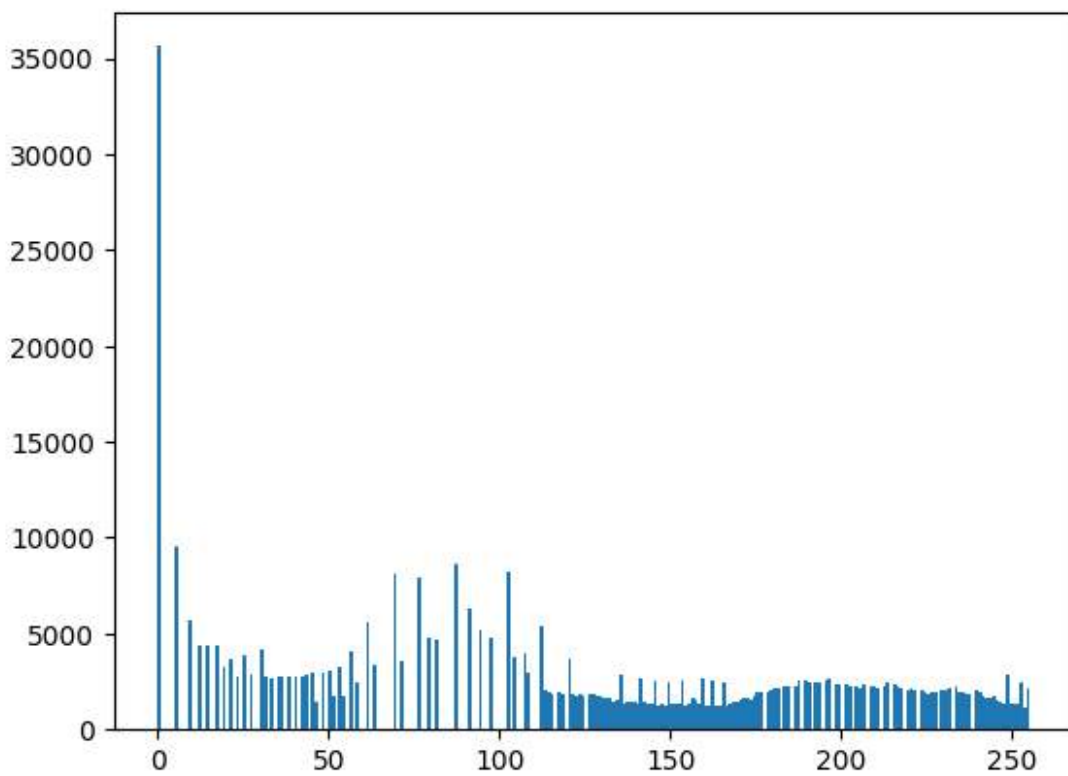
```



```
# Extra: try to visualize the histogram of the image after histogram  
equalization  
# Before histogram equalization  
plt.hist(im4.ravel(),bins = 256, range = [0, 256])  
plt.show()
```



```
# After histogram equalization  
plt.hist(im5.ravel(),bins = 256, range = [0, 256])  
plt.show()
```



##Step5: Fourier Transform of Gray Images [FFT](#)

A fourier transform is used to find the frequency domain of an image. You can consider an image as a signal which is sampled in two directions. So taking a fourier transform in both X and Y directions gives you the frequency representation of image. For the sinusoidal signal, if the amplitude varies so fast in short time, you can say it is a high frequency signal. If it varies slowly, it is a low frequency signal. Edges and noises are high frequency contents in an image because they change drastically in images.

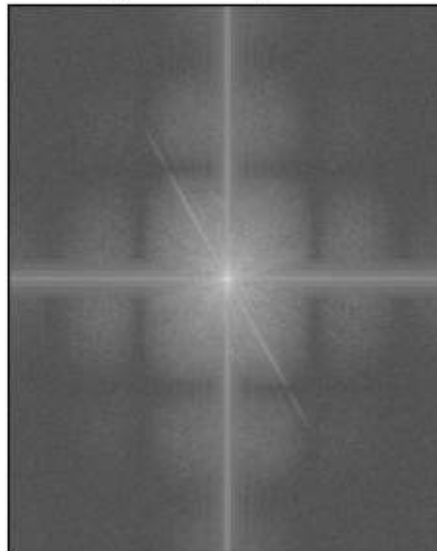
```
# Blur the grayscale image by a Guassian filter with kernel size of 10
imBlur = cv.blur(gray_image,(5,5))
# Transform the image to frequency domain
f = np.fft.fft2(imBlur)
# Bring the zero-frequency component to the center
fshift = np.fft.fftshift(f)
magnitude_spectrum = 30*np.log(np.abs(fshift))

plt.subplot(121),plt.imshow(imBlur, cmap = 'gray')
plt.title('Input Image'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(magnitude_spectrum, cmap = 'gray')
plt.title('Magnitude Spectrum'), plt.xticks([]), plt.yticks([])
plt.show()
```

Input Image



Magnitude Spectrum



Step6: Finding Edges by Highpass Filtering in FFT

This section demonstrates conducting a high pass filter to remove the low frequency component, resulting in a sharpened image which contains the edges

```
rows, cols = imBlur.shape
crow,ccol = round(rows/2) , round(cols/2)
# remove low frequencies with a rectangle size of 10
fshift[crow-10:crow+10, ccol-10:ccol+10] = 0
f_ishift = np.fft.ifftshift(fshift)
img_back = np.fft.ifft2(f_ishift)
```



```
img_back = np.abs(img_back)

plt.figure(figsize=(20, 20))
plt.subplot(131),plt.imshow(imBlur, cmap = 'gray')
plt.title('Input Image'), plt.xticks([]), plt.yticks([])
plt.subplot(132),plt.imshow(img_back, cmap = 'gray')
plt.title('Image after HPF'), plt.xticks([]), plt.yticks([])
plt.subplot(133),plt.imshow(img_back)
plt.title('Result in JET'), plt.xticks([]), plt.yticks([])
plt.show()
```



4. Object Detection (30 minutes)

Object detection is a computer vision technique that allows us to identify and locate objects in an image or video.

Significance:

1. **Real-world applications:** From face recognition in security cameras to counting objects on production lines, object detection powers numerous practical applications.
 2. **Augmentation of Image Classification:** While image classification gives a label to an entire image, object detection provides labels and locations for multiple objects within the image.
 3. **Foundation for Advanced Techniques:** Object detection serves as a base for other advanced computer vision tasks such as object tracking, image segmentation, and pose estimation.
-

Overview of Popular Object Detection Algorithms (15 minutes)

1. **Haar Cascades:**
 - **Principle:** Based on the observation that parts of an object often have contrasting pixel intensities. Haar-like features represent these intensity differences.
 - **Training:** Typically requires positive (images with the object) and negative (images without the object) samples. The AdaBoost algorithm is commonly used.
 - **Applications:** Widely used for real-time face detection.

- **Pros:** Fast and suitable for real-time applications.
 - **Cons:** Might not work well with different orientations, complex backgrounds, or non-frontal faces.
2. **YOLO (You Only Look Once):**
 - **Principle:** Divides the image into a grid, and for each grid cell predicts bounding boxes and class probabilities. Merges bounding box predictions that correspond to the same object.
 - **Training:** End-to-end learning using convolutional neural networks (CNNs).
 - **Applications:** Real-time object detection for various objects in varied scenarios.
 - **Pros:** Fast and accurate; can detect multiple objects of different classes in one pass.
 - **Cons:** Might struggle with small objects and overlapping objects.
 3. **SSD (Single Shot MultiBox Detector):**
 - **Principle:** Similar to YOLO but uses multiple feature maps from different layers of the network to predict objects at different scales.
 - **Training:** End-to-end learning with CNNs.
 - **Applications:** Widely used in scenarios where varying object sizes are present.
 - **Pros:** Handles multiple object sizes; relatively fast and accurate.
 - **Cons:** Trade-off between speed and accuracy; not as fast as YOLO but often more accurate.

5. Image Classification (30 minutes)

Image classification is a fundamental task in computer vision where an algorithm is trained to predict the class (or classes) of an image from a predefined set.

Applications:

1. **Medical Imaging:** Identify diseases or abnormalities in X-rays, MRIs, and other medical images.
 2. **Autonomous Vehicles:** Recognizing traffic signs, pedestrians, and other vehicles for safe navigation.
 3. **Agriculture:** Identifying unhealthy plants or predicting the type of crops in satellite images.
 4. **Retail:** Automated checkout systems where the items are recognized through cameras.
 5. **Social Media Platforms:** Automatic tagging of photos or content recommendation based on image content.
-

Introduction to Popular Deep Learning Models for Image Classification

Convolutional Neural Networks (CNNs):

- **Core Concept:** CNNs are designed to automatically and adaptively learn spatial hierarchies of features from images. They have been highly successful in areas such as image recognition and classification.
- **Layers:**
 - a. **Convolutional Layer:** Responsible for the convolutional operation where specific features in the image are detected.

- b. **Pooling Layer:** Reduces spatial size and computational complexity by down-sampling.
 - c. **Fully Connected Layer:** A traditional multi-layer perceptron that uses a softmax function in the output layer.
- **Notable Architectures:**
 - a. **LeNet:** One of the first convolutional architectures designed for handwritten digit recognition.
 - b. **AlexNet:** Significantly deeper than previous models and introduced the use of GPU for training.
 - c. **VGGNet:** Known for its deep layers (up to 19).
 - d. **ResNet:** Introduced skip connections to allow deeper networks by mitigating the vanishing gradient problem.

Conclusion and Wrap-up

As we conclude our exploration into the world of Computer Vision, it's essential to reflect upon the transformative power this domain holds. The ability to equip machines with the gift of 'sight' and 'perception' is no less than revolutionary. Here's a brief wrap-up of what we discussed and the potential future of computer vision:

1. **The Journey:** We began our journey with foundational concepts, understanding how computers interpret images and videos. From basic image processing techniques to the more advanced realms of object detection and image classification, our exploration has been both extensive and profound.
2. **Applications:** Throughout our discussions, we've seen how computer vision powers numerous real-world applications. From medical diagnostics to self-driving cars, and from augmented reality to automated surveillance, computer vision is ubiquitously transforming industries.
3. **Deep Learning & CNNs:** One of the significant takeaways is the symbiotic relationship between computer vision and deep learning, particularly Convolutional Neural Networks (CNNs). These neural structures have catapulted the accuracy and efficiency of vision-based tasks, driving much of the current advancements.
4. **Challenges & Future Directions:**
 - **Ethical Considerations:** As computer vision technologies become more pervasive, the ethical implications, especially concerning privacy and surveillance, become paramount. As practitioners, we must strive for responsible and transparent applications.
 - **Generalization:** While we've made significant progress, ensuring computer vision models generalize well across diverse scenarios remains a challenge. Research into few-shot learning, unsupervised learning, and domain adaptation holds promise in this direction.
 - **Combining Modalities:** Future advancements may lie in multi-modal systems where vision is combined with other senses like hearing (audio processing) or touch (haptic feedback) to create more holistic AI systems.

5. **Empowerment:** One of the most exciting aspects of computer vision is its democratization. With open-source libraries, online courses, and community support, anyone with an interest can embark on a journey to understand and innovate in this space.
6. **Final Thoughts:** Computer vision is more than just teaching machines to see; it's about enhancing our capabilities, understanding our world better, and creating tools that can solve complex, visually-oriented problems. The future is undeniably bright, filled with potential breakthroughs, applications, and innovations.

Computer Vision in AI. Practical Tasks

##Step1: Load the Dependencies

This section should load some required libraries used in this notebook: **numpy**, **pandas**, **cv2**, **skimage**, **PIL**, **matplotlib**

- **Numpy** is an array manipulation library, used for linear algebra, Fourier transform, and random number capabilities.
- **Pandas** is a library for data manipulation and data analysis.
- **CV2** is a library for computer vision tasks.
- **Skimage** is a library which supports image processing applications on python.
- **Matplotlib** is a library which generates figures and provides graphical user interface toolkit.

Useful intro about [OpenCV](#)

```
import numpy as np
import pandas as pd
import cv2 as cv
from google.colab.patches import cv2_imshow # for image display
from skimage import io
from PIL import Image
import matplotlib.pyplot as plt
```

##Step2: Read Image from Urls

In this step you will read images from urls, and display them using openCV, please note the difference when reading image in RGB and BGR format. The default input color channels are in BGR format for openCV.

```
# Create a list to store the urls of the images
urls =
["https://iiif.lib.ncsu.edu/iiif/0052574/full/800,/0/default.jpg",
"https://iiif.lib.ncsu.edu/iiif/0016007/full/800,/0/default.jpg",
 "https://placekitten.com/800/571"]
# Read and display the image
# loop over the image URLs, you could store several image urls in
the list

for url in urls:
    image = io.imread(url)
    image_2 = cv.cvtColor(image, cv.COLOR_BGR2RGB)
    final_frame = cv.hconcat((image, image_2))
    cv2_imshow(final_frame)
    print('\n')
```

TODO #1: Read an image from a URL and display it

Image source examples:

[Place Kitten](#) - use the base Place Kitten URL followed by a width and height separated by backslashes "/". For example, use the URL `https://placekitten.com/500/300` to fetch a cat image with a width of 500px and height of 300px.

[Google Image search](#) - search for an image. Left-click one of the returned images, then right-click on the full image, and then select "Copy Image Address".

```
## TODO: LOAD IMAGE
## url =
## myImg = io.imread(url)
## cv2_imshow(cv.cvtColor(myImg, cv.COLOR_BGR2RGB))
```

##Step3: Image Contours and Histograms

```
# Check the image matrix data type (could know the bit depth of the image)
print(image.dtype)
# Check the height of image
print(image.shape[0])
# Check the width of image
print(image.shape[1])
# Check the number of channels of the image
print(image.shape[2])
```

Generate Histogram of color image and grayscale image

More info: [Histogram](#)

```
plt.hist(image.ravel(),bins = 256, range = [0,256])
plt.show()
```

Display the histogram of R, G, B channel We could observe that the green channel has many pixels in 255, which represents the white patch in the image

```
color = ('b','g','r')
for i,col in enumerate(color):
    histr = cv.calcHist([image],[i],None,[256],[0,256])
    plt.plot(histr,color = col)
    plt.xlim([0,256])
plt.show()

gray_image = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
cv2_imshow(gray_image)

# Plot the histogram of the gray image. We could observe that the frequency of
# the image hist has decreased ~ 1/3 of the histogram of color image
plt.hist(gray_image.ravel(),bins = 256, range = [0, 256])
plt.show()
```

TODO # 2: DISPLAY THE GRAYSCALE OF YOUR COLOR IMAGE AND GENERATE HISTOGRAM

```
#myGrayImg = cv.cvtColor(myImg, cv.COLOR_BGR2GRAY)
#cv2_imshow()
```

Find image contour of the grayscale image

Method 1: Use the matplotlib. contour

More Info: [matplotlib contour](#)

```
plt.contour(gray_image, origin = "image")
```

Method 2: Use the openCV lib

More info: [Contour](#)

```
# Set threshold for the countour detection
ret, thresh = cv.threshold(gray_image, 150, 255, 0)
im2, contours, hierarchy = cv.findContours(thresh, cv.RETR_TREE,
cv.CHAIN_APPROX_SIMPLE)
cv.drawContours(image, contours, -1, (0, 255, 0), 3)
plt.imshow(image)
```

TODO # 3: Find the contour of your own image

```
# Your code
# plt.contour()
```

##Step4: Grayscale Transform and Histogram Equalization

```
# This is an inverse operation of the grayscale image, you could see
that the
# bright pixels become dark, and the dark pixels become bright
im2 = 255 - gray_image
cv2_imshow(im2)

# Another transform of the image, after adding a constant,
# all the pixels become brighter and a hazing-like effect of the
image is generated
im3 = (100.0/255)*gray_image + 100
cv2_imshow(im3)

# The lightness level of the gray_image decreases after this step
im4 = 255.0*(gray_image/255.0)**2
cv2_imshow(im4)
```

TODO #4: Try some mathematical operations on your image

```
## Implement your code here
## myGrayImgTrans = myGrayImg...
## cv2_imshow(myGrayImgTrans)
```

Histogram Equalization

In this section you will demonstrate histogram equalization on a dark image.

```

# function of histogram equalization
def histeq(im, nbr_bins = 256):
    """ Histogram equalization of a grayscale image. """
    # get the image histogram
    imhist, bins = np.histogram(im.flatten(), nbr_bins, [0, 256])
    cdf = imhist.cumsum() # cumulative distribution function
    cdf = imhist.max()*cdf/cdf.max() #normalize
    cdf_mask = np.ma.masked_equal(cdf, 0)
    cdf_mask = (cdf_mask - cdf_mask.min())*255/(cdf_mask.max() -
cdf_mask.min())
    cdf = np.ma.filled(cdf_mask,0).astype('uint8')
    return cdf[im.astype('uint8')]

# apply the function on your dark image to increase the contrast
# we could observe that the contrast of the black background has
increased
im5 = histeq(im4)
cv2_imshow(im5)

# Extra: try to visualize the histogram of the image after histogram
equalization
# Before histogram equalization
plt.hist(im4.ravel(),bins = 256, range = [0, 256])
plt.show()

# After histogram equalization
plt.hist(im5.ravel(),bins = 256, range = [0, 256])
plt.show()

```

TODO # 5: Do a histogram equalization on your grayscale image

```

## TODO: IMPLEMENT YOUR CODE HERE
## CODE INSTRUCTION
##plt.imshow()

```

##Step5: Fourier Transform of Gray Images [FFT](#)

A fourier transform is used to find the frequency domian of an image. You can consider an image as a signal which is sampled in two directions. So taking a fourier transform in both X and Y directions gives you the frequency representation of image. For the sinusoidal signal, if the amplitude varies so fast in short time, you can say it is a high frequency signal. If it varies slowly, it is a low frequency signal. Edges and noises are high frequency contents in an image because they change drastically in images.

```

# Blur the grayscale image by a Guassian filter with kernel size of
10
imBlur = cv.blur(gray_image,(5,5))
# Transform the image to frequency domain
f = np.fft.fft2(imBlur)
# Bring the zero-frequency component to the center
fshift = np.fft.fftshift(f)
magnitude_spectrum = 30*np.log(np.abs(fshift))

```



```
plt.subplot(121),plt.imshow(imBlur, cmap = 'gray')
plt.title('Input Image'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(magnitude_spectrum, cmap = 'gray')
plt.title('Magnitude Spectrum'), plt.xticks([]), plt.yticks([])
plt.show()
```

TODO # 6: Generate a fourier transform of your grayscale image

todo: Implement your code here

Step6: Finding Edges by Highpass Filtering in FFT

In this section you will use a high pass filter to remove the low frequency component, resulting in a sharpened image which contains the edges

```
rows, cols = imBlur.shape
crow,ccol = round(rows/2) , round(cols/2)
# remove low frequencies with a rectangle size of 10
fshift[crow-10:crow+10, ccol-10:ccol+10] = 0
f_ishift = np.fft.ifftshift(fshift)
img_back = np.fft.ifft2(f_ishift)
img_back = np.abs(img_back)

plt.figure(figsize=(20, 20))
plt.subplot(131),plt.imshow(imBlur, cmap = 'gray')
plt.title('Input Image'), plt.xticks([]), plt.yticks([])
plt.subplot(132),plt.imshow(img_back, cmap = 'gray')
plt.title('Image after HPF'), plt.xticks([]), plt.yticks([])
plt.subplot(133),plt.imshow(img_back)
plt.title('Result in JET'), plt.xticks([]), plt.yticks([])
plt.show()
```

TODO # 7: Use High Pass Filtering

todo: Implement your code here

```
!pip install git+https://github.com/mehalyna/cooltest.git
```

```
from cooltest.test_cool_5 import *
```

Pass

Practical Task #1. Image Color Detection

Description: Write a Python program that takes an image as input and detects the dominant color in the image. You can use a library like OpenCV to read the image and perform color analysis. Your program should display the detected color along with the percentage of its presence in the image.

Steps:

1. Import the necessary libraries (OpenCV, numpy).
2. Read the input image using OpenCV.

3. Convert the image from BGR to HSV color space (for better color analysis).
4. Apply a clustering algorithm (such as K-Means) to identify the dominant color.
5. Calculate the percentage of the dominant color's presence in the image.
6. Display the detected color and its percentage.

```
from google.colab import drive
drive.mount('/content/drive')
%cd /content/drive/My Drive/Colab Notebooks/SoftServe AI/

import cv2
import numpy as np
from sklearn.cluster import KMeans # Import KMeans class from
scikit-learn

@test_detect_color_task
def detect_dominant_color(image_path):
    """
    Detects the dominant color in an image and calculates its
    presence percentage.

    Args:
        image_path (str): Path to the input image file.

    Returns:
        dominant_color (tuple): RGB values of the detected dominant
        color.
        percentage (float): Percentage of the dominant color's
        presence in the image.
    """
    # Read the image

    # Convert image from BGR to HSV color space

    # Flatten the HSV image into a 1D array

    # Apply K-Means clustering

    # Number of clusters (you can adjust this)

    # Get the dominant color cluster

    # Convert dominant color from HSV to RGB

    # Calculate the percentage of dominant color's presence

    return dominant_color, percentage

# Path to the input image

# Call the function and get the results
dominant_color, percentage = detect_dominant_color(image_path)

# Print the results
```

```
print(f"Dominant Color: {dominant_color}")  
print(f"Percentage: {percentage:.2f}%")
```

Topic: Genetic Algorithms in AI

Duration: 2 hours

Agenda:

1. Introduction to Genetic Algorithms
 - a. Understanding the basics of Genetic Algorithms (GAs)
 - b. Inspiration from natural evolution and genetics
 - c. Components of a Genetic Algorithm: Representation, selection, crossover, and mutation
2. Genetic Operators
 - a. Selection methods in Genetic Algorithms
 - Fitness proportionate selection (Roulette wheel selection)
 - Tournament selection
 - b. Crossover (Recombination) techniques
 - One-point crossover
 - Two-point crossover
 - Uniform crossover
 - c. Mutation operators and their role in introducing diversity
3. Genetic Algorithm Workflow
 - a. Initialization: Creating an initial population
 - b. Evaluation: Calculating fitness for each individual
 - c. Selection: Choosing individuals for reproduction
 - d. Crossover and Mutation: Creating new offspring
 - e. Replacement: Updating the population for the next generation
4. Hands-on Exercise
 - a. Implementing a simple Genetic Algorithm in Python
 - b. Applying the Genetic Algorithm to solve an optimization problem
5. Advantages and Applications of Genetic Algorithms
 - a. Advantages and strengths of Genetic Algorithms
 - b. Applications of GAs in optimization problems
 - c. GAs in solving complex search problems

- d. Genetic programming and automatic code generation
- 6. Real-world Applications
 - a. Real-world use cases of Genetic Algorithms
- 7. Q&A and Wrap-up
 - a. Answering participant questions and clarifications
 - b. Summary of key concepts covered in the course
 - c. Encouraging further exploration of Genetic Algorithms and AI applications

The theory of evolution suggests that organisms evolve through reproduction by producing children of mixed genes from their parents; and given the fitness of these individuals in their environment, stronger individuals have a higher likelihood of survival.

The **Knapsack Problem** is one of the famous problems in optimization, specifically in combinatoric optimization. The motivation of this problem comes where someone needs to maximize his knapsack capacity — hence the name — with the most valuable items possible. There are many approaches to solve this problem, but in this class, we will look at an example to solve this problem using the Genetic Algorithm approach in Python.

Problem Definition:

The *zero-one knapsack problem* belongs to the category of combinatorial optimization problems. Combinatorial optimization problems typically involve the selection of an optimal solution from a finite set of solutions. An exhaustive search for these problems is usually not feasible as it may be computationally expensive and time-consuming.

The **problem's primary objective** is to fill a knapsack of fixed capacity with the most profitable items from a set of items. Each item has an associated weight, and thus, selecting a specific item consumes its associated weight from the knapsack's fixed capacity. This type of problem is analogous to a resource allocation task where one may include or exclude a non-divisible job in a budget-bound or time-bound queue.

The Zero-One here denotes the boolean status of whether an item is excluded or included respectively. A basic condition for this specific problem statement is that the items are indivisible and cannot be included in parts.

1. Genetic Algorithm Concepts

Genetic Algorithm is one of the optimization algorithms based on the evolution concept by natural selection. As proposed by Charles Darwin, evolution by natural selection is the mechanism on how many varieties of living things will adapt to the environment to survive through two principles: natural selection and mutation. Based on that concept, the Genetic Algorithm's goal is to gain the optimal solutions of the objective function by selecting the best or fittest solution alongside the rare and random mutation occurrence.

Basic terms of the genetic algorithm

The population of individuals are maintained within **Search Space**. Each individual represents a solution in search space for given problem. Each individual is coded as a finite length vector of components - **Chromosome**. These variable components are analogous to **Genes** or

Genome. Thus a chromosome (individual) is composed of several genes (variable components).

A **Fitness Score** is given to each individual which shows the ability of an individual to "compete". The individual having optimal fitness score (or near optimal) are sought.

The GAs maintains the **Population** of n individuals (chromosome/solutions) along with their fitness scores. The individuals having better fitness scores are given more chance to reproduce than others. The individuals with better fitness scores are selected who mate and produce better **offspring** by combining chromosomes of parents.

Each new generation has on average more "better genes" than the individual (solution) of previous generations. Thus each new generations have better "partial solutions" than previous generations.

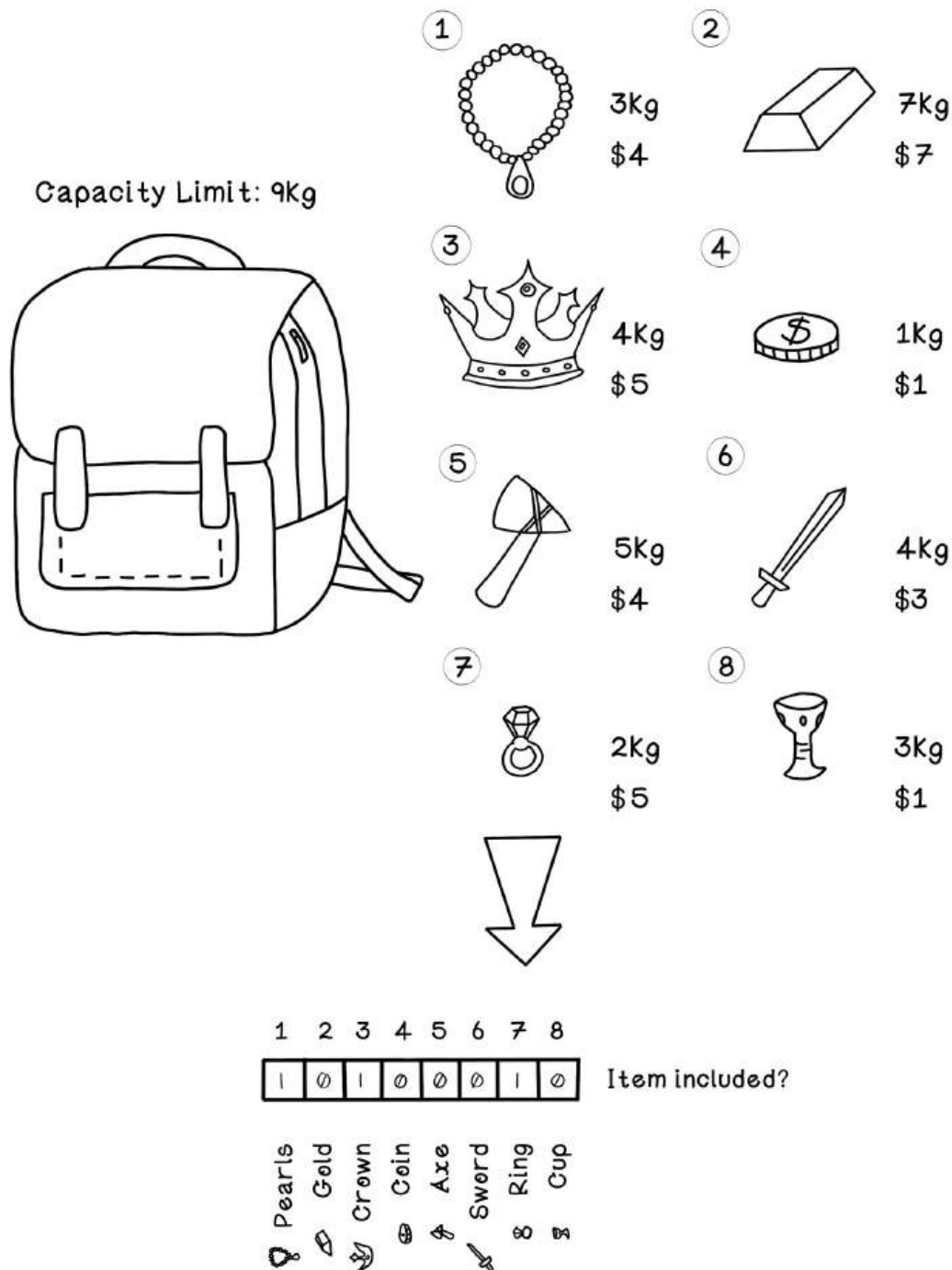
The Genetic Algorithm evolves the generation using following operators:

- 1) **Selection Operator:** The idea is to give preference to the individuals with good fitness scores and allow them to pass their genes to successive generations.
- 2) **Crossover Operator:** This represents mating between individuals. Two individuals are selected using selection operator and crossover sites are chosen randomly. Then the genes at these crossover sites are exchanged thus creating a completely new individual (offspring).
- 3) **Mutation Operator:** The key idea is to insert random genes in offspring to maintain the diversity in the population to avoid premature convergence.

The **Genetic Algorithm** itself can be explained as follows.

1. Initialize the data and/or the function that we will optimize.
2. Initialize the population size, maximum iteration number (the number of generations), crossover probability, mutation probability, and the number of elitism (the best or fittest individual that won't undergo mutation).
3. From the population, select two individuals, then perform crossover between two individuals with probability p .
4. Then, perform mutation between two individuals with probability p (usually the probability of mutation are really low).
5. Repeat steps 3 and 4 until all individuals in one generation are trained. These all individuals will be used for training the next generation until the number of generations is reaching the limit.

To give you a better understanding of the genetic algorithm, let's jump to the case study where we will use the genetic algorithm to solve the knapsack problem in Python.



Step 1: Initialization

The population size for each generation is usually problem-specific; even then, it usually contains several thousand solution candidates. The chromosomes or the genotype of the individuals in the initial population are typically generated at random. In some cases, these chromosomes may also result from a seed value, when the solution of the problem is likely to be found in a particular region of the search space.

The genotype of an individual in a generation can be represented as an array of values, indicating its properties. Typically, an array of bits is used for this purpose.

The initial population is stochastically generated and represented as a matrix, with each row as an individual genotype sequence.

```
# At first we need to import necessary Python modules

import numpy as np
import random

from datetime import datetime

# ... and initialize parameters

n_objects = 20
max_weight = 3

n_population = 100

mutation_rate = 0.3
```

Suppose that you want to go hiking with your friends, and you have the items that you can use for hiking, with the weight (in kilograms) and survival point of each item, respectively, as follows.

Few slides about [GA general approach for Knapsack Problem](#) before we'll start

```
# Generating a list of weights and values representing each item
weight_value = [[x,y] for x,y in
zip(np.random.randint(0,10,n_objects)/10,np.random.randint(0,100,n_o
bjects))]
object_list = np.array(['Water', 'Food', 'Pants', 'Socks', 'Boots',
'Shirts', 'Coat', 'Blanket', 'Laptop', 'Sleeping Bag', \
                        'Cellphone', 'Book', 'Gloves', 'Towel',
'Sunscream', 'Glasses', 'Fork', 'Knife', 'Matches', 'Umbrella'])
objects_dict = { x:y for x,y in zip(object_list,weight_value)}

objects_dict

{'Water': [0.8, 87],
 'Food': [0.5, 19],
 'Pants': [0.6, 95],
 'Socks': [0.7, 57],
 'Boots': [0.0, 74],
 'Shirts': [0.5, 64],
 'Coat': [0.2, 54],
 'Blanket': [0.3, 68],
 'Laptop': [0.3, 72],
 'Sleeping Bag': [0.1, 64],
 'Cellphone': [0.3, 12],
 'Book': [0.5, 68],
 'Gloves': [0.7, 43],
 'Towel': [0.0, 88],
 'Sunscream': [0.7, 32],
 'Glasses': [0.4, 50],
 'Fork': [0.5, 74],
```



```

'Knife': [0.2, 47],
'Matches': [0.9, 67],
'Umbrella': [0.6, 2]}

def get_current_weight_value(objects_list, objects_dict):
    weight = 0
    value = 0
    for o in objects_list:
        o = objects_dict[o]
        weight += o[0]
        value += o[1]
    return weight, value

def s_print(obj_dict):
    for item in set(obj_dict):
        print(item)

```

Create the first population set

We randomly shuffle the items N times where N=population_size

```

# First step: Create the first population set
def fit_in_knapsack(objects_list, max_weight, objects_dict):
    r = []
    for o in objects_list:
        r.append(o)
        # Check if count(item) > 1
        for item in r:
            flag = r.count(item) > 1
            weight, value = get_current_weight_value(r, objects_dict)
            if (weight > max_weight) or flag:
                r.pop()
        return r
    return r

def genesis(object_list, n_population, max_weight, objects_dict):
    population_set = []
    for i in range(n_population):
        #Randomly generating a new solution
        sol_i = object_list[np.random.choice(list(range(n_objects)),
n_objects, replace=False)]
        sol_i = fit_in_knapsack(sol_i, max_weight, objects_dict)
        population_set.append(sol_i)
    return np.array(population_set, dtype=object)

population_set = genesis(object_list, n_population, max_weight,
objects_dict)
population_set
array([list(['Sleeping Bag', 'Shirts', 'Book', 'Blanket', 'Matches',
'Socks']),
      list(['Sleeping Bag', 'Umbrella', 'Water', 'Fork',

```

```

'Sunscream']],
    list(['Coat', 'Book', 'Sunscream', 'Glasses', 'Umbrella',
'Knife'])),
    list(['Water', 'Matches', 'Gloves', 'Food', 'Towel']),
    list(['Blanket', 'Food', 'Water', 'Socks', 'Glasses',
'Sleeping Bag'])),
    list(['Blanket', 'Laptop', 'Water', 'Shirts', 'Matches']),
    list(['Glasses', 'Towel', 'Sleeping Bag', 'Gloves',
'Umbrella', 'Water'])),
    list(['Gloves', 'Knife', 'Umbrella', 'Coat', 'Matches']),
    list(['Sunscream', 'Boots', 'Sleeping Bag', 'Water',
'Shirts', 'Socks'])),
    list(['Knife', 'Boots', 'Sleeping Bag', 'Water', 'Towel',
'Coat', 'Blanket', 'Gloves', 'Shirts'])),
    list(['Book', 'Cellphone', 'Gloves', 'Blanket', 'Laptop',
'Towel', 'Socks', 'Coat'])),
    list(['Book', 'Shirts', 'Pants', 'Matches']),
    list(['Boots', 'Book', 'Coat', 'Knife', 'Sleeping Bag',
'Umbrella', 'Fork', 'Pants'])),
    list(['Sunscream', 'Water', 'Shirts', 'Gloves', 'Boots',
'Towel', 'Blanket'])),
    list(['Shirts', 'Umbrella', 'Sleeping Bag', 'Fork', 'Laptop',
'Glasses', 'Blanket'])),
    list(['Food', 'Shirts', 'Water', 'Sunscream']),
    list(['Pants', 'Shirts', 'Blanket', 'Cellphone', 'Fork',
'Knife', 'Glasses'])),
    list(['Socks', 'Cellphone', 'Food', 'Glasses', 'Knife',
'Sleeping Bag', 'Book'])),
    list(['Socks', 'Sunscream', 'Glasses', 'Cellphone',
'Gloves'])),
    list(['Blanket', 'Glasses', 'Book', 'Umbrella', 'Sleeping
Bag', 'Pants'])),
    list(['Gloves', 'Shirts', 'Boots', 'Coat', 'Food', 'Glasses',
'Fork'])),
    list(['Coat', 'Boots', 'Sleeping Bag', 'Food', 'Gloves',
'Cellphone', 'Glasses', 'Pants'])),
    list(['Towel', 'Pants', 'Gloves', 'Sunscream', 'Book']),
    list(['Fork', 'Book', 'Matches', 'Coat', 'Sleeping Bag',
'Blanket', 'Boots', 'Shirts'])),
    list(['Shirts', 'Blanket', 'Pants', 'Gloves', 'Book',
'Knife'])),
    list(['Laptop', 'Food', 'Book', 'Matches', 'Fork']),
    list(['Gloves', 'Laptop', 'Boots', 'Book', 'Pants', 'Sleeping
Bag'])),
    list(['Fork', 'Water', 'Pants', 'Umbrella']),
    list(['Gloves', 'Book', 'Umbrella', 'Shirts', 'Fork',
'Knife', 'Towel', 'Boots'])),
    list(['Shirts', 'Glasses', 'Water', 'Laptop', 'Socks',
'Sleeping Bag'])),
    list(['Pants', 'Towel', 'Matches', 'Coat', 'Shirts',
'Cellphone', 'Boots', 'Knife'])),
    list(['Shirts', 'Cellphone', 'Boots', 'Blanket', 'Coat',
'Water', 'Towel', 'Sleeping Bag', 'Food']),

```

```

list(['Fork', 'Blanket', 'Shirts', 'Umbrella', 'Matches',
'Sleeping Bag']),
list(['Glasses', 'Fork', 'Umbrella', 'Socks', 'Shirts',
'Towel', 'Coat']),
list(['Umbrella', 'Socks', 'Glasses', 'Food', 'Book']),
list(['Water', 'Matches', 'Gloves', 'Sleeping Bag']),
list(['Coat', 'Sleeping Bag', 'Cellphone', 'Shirts',
'Sunscream', 'Gloves', 'Blanket']),
list(['Fork', 'Laptop', 'Pants', 'Water', 'Coat']),
list(['Gloves', 'Glasses', 'Towel', 'Cellphone', 'Shirts',
'Water']),
list(['Sunscream', 'Socks', 'Sleeping Bag', 'Knife',
'Blanket', 'Gloves', 'Laptop']),
list(['Pants', 'Sunscream', 'Shirts', 'Socks', 'Food']),
list(['Sleeping Bag', 'Cellphone', 'Socks', 'Shirts',
'Gloves', 'Blanket']),
list(['Matches', 'Boots', 'Glasses', 'Laptop', 'Sleeping
Bag', 'Cellphone', 'Book', 'Knife']),
list(['Food', 'Gloves', 'Umbrella', 'Shirts', 'Cellphone']),
list(['Food', 'Cellphone', 'Water', 'Sleeping Bag', 'Coat',
'Umbrella', 'Knife', 'Boots']),
list(['Cellphone', 'Boots', 'Food', 'Glasses', 'Water',
'Laptop', 'Socks']),
list(['Blanket', 'Food', 'Coat', 'Water', 'Pants', 'Sleeping
Bag']),
list(['Cellphone', 'Gloves', 'Towel', 'Sleeping Bag',
'Blanket', 'Laptop', 'Shirts', 'Glasses', 'Coat']),
list(['Fork', 'Socks', 'Coat', 'Water', 'Towel', 'Food']),
list(['Food', 'Cellphone', 'Matches', 'Knife', 'Shirts']),
list(['Water', 'Boots', 'Glasses', 'Cellphone', 'Umbrella',
'Laptop']),
list(['Socks', 'Shirts', 'Umbrella', 'Matches']),
list(['Gloves', 'Sleeping Bag', 'Book', 'Matches', 'Pants']),
list(['Umbrella', 'Sunscream', 'Cellphone', 'Pants',
'Socks']),
list(['Food', 'Cellphone', 'Laptop', 'Blanket', 'Gloves',
'Sunscream', 'Sleeping Bag']),
list(['Coat', 'Gloves', 'Glasses', 'Food', 'Towel', 'Sleeping
Bag', 'Water']),
list(['Blanket', 'Boots', 'Knife', 'Matches', 'Umbrella',
'Food', 'Glasses', 'Sleeping Bag']),
list(['Coat', 'Boots', 'Shirts', 'Matches', 'Sleeping Bag',
'Food', 'Sunscream']),
list(['Knife', 'Blanket', 'Matches', 'Book', 'Coat',
'Glasses']),
list(['Socks', 'Blanket', 'Food', 'Pants', 'Glasses']),
list(['Boots', 'Sleeping Bag', 'Cellphone', 'Towel', 'Water',
'Blanket', 'Food', 'Umbrella']),
list(['Fork', 'Sleeping Bag', 'Blanket', 'Knife', 'Shirts',
'Glasses', 'Food', 'Book', 'Boots']),
list(['Shirts', 'Fork', 'Coat', 'Gloves', 'Knife', 'Towel',
'Cellphone', 'Laptop']),
list(['Knife', 'Food', 'Sunscream', 'Matches', 'Socks']),

```

```

    list(['Cellphone', 'Umbrella', 'Matches', 'Socks', 'Laptop',
'Coat']),
    list(['Cellphone', 'Sleeping Bag', 'Glasses', 'Blanket',
'Knife', 'Food', 'Towel', 'Shirts', 'Book', 'Boots']),
    list(['Boots', 'Water', 'Socks', 'Knife', 'Food', 'Sleeping
Bag', 'Coat', 'Laptop']),
    list(['Food', 'Shirts', 'Sunscream', 'Book', 'Fork',
'Sleeping Bag']),
    list(['Sleeping Bag', 'Pants', 'Fork', 'Water', 'Umbrella',
'Boots']),
    list(['Sunscream', 'Laptop', 'Book', 'Blanket', 'Umbrella',
'Coat']),
    list(['Laptop', 'Sleeping Bag', 'Shirts', 'Boots', 'Fork',
'Water', 'Sunscream']),
    list(['Sunscream', 'Knife', 'Pants', 'Blanket', 'Gloves',
'Fork']),
    list(['Fork', 'Cellphone', 'Blanket', 'Pants', 'Shirts',
'Umbrella']),
    list(['Umbrella', 'Boots', 'Socks', 'Sleeping Bag', 'Pants',
'Fork']),
    list(['Sleeping Bag', 'Pants', 'Knife', 'Book', 'Umbrella',
'Shirts', 'Cellphone']),
    list(['Pants', 'Boots', 'Socks', 'Coat', 'Sunscream',
'Water']),
    list(['Pants', 'Socks', 'Boots', 'Knife', 'Sleeping Bag',
'Water', 'Blanket', 'Towel']),
    list(['Gloves', 'Coat', 'Pants', 'Fork', 'Knife',
'Sunscream']),
    list(['Cellphone', 'Boots', 'Towel', 'Laptop', 'Water',
'Knife', 'Sunscream', 'Coat', 'Shirts']),
    list(['Knife', 'Book', 'Water', 'Pants', 'Socks']),
    list(['Fork', 'Blanket', 'Laptop', 'Shirts', 'Book', 'Boots',
'Towel', 'Gloves']),
    list(['Water', 'Sunscream', 'Blanket', 'Coat', 'Socks']),
    list(['Towel', 'Fork', 'Book', 'Boots', 'Shirts', 'Umbrella',
'Cellphone']),
    list(['Pants', 'Fork', 'Matches', 'Laptop', 'Gloves']),
    list(['Glasses', 'Coat', 'Sleeping Bag', 'Book', 'Boots',
'Matches', 'Knife', 'Socks']),
    list(['Cellphone', 'Laptop', 'Umbrella', 'Water', 'Socks']),
    list(['Laptop', 'Sleeping Bag', 'Blanket', 'Book', 'Shirts',
'Water', 'Coat']),
    list(['Coat', 'Boots', 'Umbrella', 'Fork', 'Sleeping Bag',
'Water', 'Book']),
    list(['Blanket', 'Matches', 'Pants', 'Water', 'Knife']),
    list(['Coat', 'Knife', 'Socks', 'Water', 'Gloves']),
    list(['Pants', 'Gloves', 'Sunscream', 'Laptop', 'Knife',
'Book']),
    list(['Fork', 'Water', 'Pants', 'Towel', 'Matches', 'Coat']),
    list(['Shirts', 'Matches', 'Laptop', 'Book', 'Gloves']),
    list(['Gloves', 'Blanket', 'Fork', 'Laptop', 'Glasses',
'Socks']),
    list(['Knife', 'Shirts', 'Water', 'Towel', 'Food', 'Laptop',

```

```
'Sunscream']],
    list(['Glasses', 'Cellphone', 'Towel', 'Water', 'Pants',
'Coat'])),
    list(['Umbrella', 'Gloves', 'Book', 'Coat', 'Water'])),
    list(['Socks', 'Matches', 'Water', 'Boots', 'Blanket'])),
    list(['Socks', 'Book', 'Coat', 'Sunscream', 'Blanket',
'Shirts'])),
    list(['Sunscream', 'Shirts', 'Knife', 'Water', 'Boots',
'Umbrella'])]],
    dtype=object)
```

Now it's time to start the process of **Natural Selection**

Step 2: Selection

From each generation, a few individuals are selected based on their fitness scores. These fitness scores are calculated for each individual based on an objective function. This function is based on the problem statement for which the Genetic Algorithm is being used and indicates an individual's quality.

This fitness score of an individual is an indication of how suitable the respective candidate is for the problem. For each successive generation, a portion of the current population with relatively higher fitness scores is allowed to move to the next iteration. Various schemes such as tournament selection, rank-based selection, roulette wheel selection, elitism selection, etc may be used in the selection stage.

Fitness Function: The fitness function that we use here takes input from the chromosome sequence and the values of the individual items. It then returns the profit by calculating the dot product of the two input vectors.

Parent Selection: For this, we may use the approach of deterministic tournament selection. In this method, we randomly select two individuals from the population and choose the one with higher fitness. We may observe cases where both individuals have low fitness scores, but they still might contribute to choosing the correct genetic sequence. This type of parent selection results in 50% of the population moving to the next stage. To increase the preference for higher fitness, we may include more than two individuals for consideration in the tournament.

Evaluate solutions fitness

The solutions are defined so that the first element on the list is the first item to take, then the second, etc. The fitness function needs to compute the total weight of the backpack.

```
def get_all_fitnes(population_set, objects_dict):
    fitnes_list = np.zeros(n_population)

    #Looping over all solutions computing the fitness for each
    solution
    for i in range(n_population):
        _, fitnes_list[i] =
get_current_weight_value(population_set[i], objects_dict)

    return fitnes_list
```

```

fitnes_list = get_all_fitnes(population_set,objects_dict)
fitnes_list

array([388., 259., 253., 304., 345., 358., 334., 213., 378., 589.,
462.,
      294., 478., 456., 394., 202., 410., 317., 194., 347., 378.,
411.,
      326., 533., 385., 300., 416., 258., 460., 394., 501., 530.,
339.,
      389., 196., 261., 337., 382., 344., 383., 267., 308., 454.,
140.,
      359., 371., 387., 515., 379., 209., 297., 190., 337., 198.,
310.,
      405., 391., 374., 354., 289., 414., 528., 454., 222., 264.,
554.,
      474., 321., 396., 296., 467., 359., 315., 366., 352., 399.,
580.,
      345., 530., 354., 551., 298., 382., 351., 481., 230., 477.,
423.,
      364., 288., 357., 465., 314., 364., 409., 386., 254., 353.,
343.,
      306.] )

```

Progenitors selection

We will select a new set of progenitors using the *Roulette Wheel Selection*. Generates a list of progenitor pairs where $N = \text{len}(\text{population\_set})$ but at each position there are two solutions to merge

```

def progenitor_selection(population_set,fitnes_list):
    total_fit = fitnes_list.sum()
    prob_list = fitnes_list/total_fit

    #Notice there is not (of False) a chance that a progenitor.
    #mates with oneself
    progenitor_list_a =
np.random.choice(list(range(len(population_set))),
len(population_set),p=prob_list, replace=False)
    progenitor_list_b =
np.random.choice(list(range(len(population_set))),
len(population_set),p=prob_list, replace=False)

    progenitor_list_a = population_set[progenitor_list_a]
    progenitor_list_b = population_set[progenitor_list_b]

    return np.array([progenitor_list_a,progenitor_list_b])

progenitor_list = progenitor_selection(population_set,fitnes_list)
progenitor_list[0][2]

['Boots',
 'Sleeping Bag',

```

```
'Cellphone',  
'Towel',  
'Water',  
'Blanket',  
'Food',  
'Umbrella']
```

Step 3: Crossover (also, called Recombination)

This is an essential phase of the Algorithm and involves the creation of a new generation from the current population of the selected individuals. Each individual in the new generation is a result of a crossover operation on a pair of “parent” solutions from the pool of selected individuals.

The new individual created contains traits from both parents based on the Crossover scheme used. The crossover operation is applied over selected individuals using one of the schemes mentioned previously. This results in a completely new population (also called as Children Population).

Mating

For each pair of parents we'll generate an offspring pair. Since we cannot repeat items what we'll do is copy a random chunk from one progenitor and fill the blanks with the other progenitor.

```
def mate_progenitors(prog_a, prog_b, max_weight, objects_dict):  
    offspring = []  
  
    for i in zip(prog_a, prog_b):  
        offspring.extend(i)  
        offspring = list(dict.fromkeys(offspring)) #Removing  
duplicates  
        offspring = fit_in_knapsack(offspring, max_weight,  
objects_dict)  
  
    return offspring  
  
def mate_population(progenitor_list, max_weight, objects_dict):  
    new_population_set = []  
    for i in range(progenitor_list.shape[1]):  
        prog_a, prog_b = progenitor_list[0][i], progenitor_list[1]  
[i]  
        offspring = mate_progenitors(prog_a, prog_b, max_weight,  
objects_dict)  
        new_population_set.append(offspring)  
  
    return new_population_set  
  
new_population_set = mate_population(progenitor_list, max_weight,  
objects_dict)  
new_population_set[0]
```

```
['Sleeping Bag', 'Fork', 'Pants', 'Knife', 'Blanket', 'Book', 'Umbrella']
```

The next step of evolution is **mutation**

Step 4: Mutation

This operation is used to create more genetic diversity in a generation. It alters one or more values from the chromosome, resulting in a completely new genome. This often helps avoid premature convergence.

This is analogous to biological mutation and alters the chromosomes of the children using one or more schemes such as bit-flip mutation, swap mutation, scramble mutation, etc. In simple terms, mutation may be defined as a small random tweak in the chromosomes, to get a new solution.

Now for each element of the new population we add a random chance of swapping

```
def mutate_offspring(offspring, max_weight, object_list,
objects_dict):
    for mutation_number in range(int(len(offspring)*mutation_rate)):
        a = np.random.randint(0, len(object_list))
        b = np.random.randint(0, len(offspring))

        offspring.insert(b, object_list[a])

        offspring = fit_in_knapsack(offspring, max_weight, objects_dict)

    return offspring

def mutate_population(new_population_set, max_weight, object_list,
objects_dict):
    mutated_pop = []
    for offspring in new_population_set:
        mutated_pop.append(mutate_offspring(offspring, max_weight,
object_list, objects_dict))
    return mutated_pop

mutated_pop = mutate_population(new_population_set,
max_weight, object_list, objects_dict)
mutated_pop[0]
```

```
['Sleeping Bag', 'Fork', 'Shirts', 'Pants', 'Umbrella', 'Gloves']
```

Step 5a: Iteration

The above steps from 2 to 4 are repeated with the newer generations until the Termination Conditions are met.

Step 5b: Termination

The loop is terminated when either of the following conditions is met:

- A solution is obtained that satisfies a Threshold fitness.
- The maximum number of generations specified has been processed.
- The allocated computation budget is consumed.
- The best fitness for the new generation does not show any significant improvement and seems to have converged at a point.

To select the stopping criteria we'll need to create a loop to stop first. Then we'll set it to loop at 1000 iterations.

```
best_solution = [-1,-np.inf,np.array([], dtype=object)]
for i in range(10000):
    if i%100==0: print(i, fitness_list.min(), fitness_list.mean(),
datetime.now().strftime("%d/%m/%y %H:%M"))
    fitness_list = get_all_fitness(mutated_pop,objects_dict)

    #Saving the best solution
    if fitness_list.max() > best_solution[1]:
        best_solution[0] = i
        best_solution[1] = fitness_list.max()
        best_solution[2] = np.array(mutated_pop, dtype=object)
[fitness_list.max() == fitness_list]

    progenitor_list =
progenitor_selection(population_set,fitness_list)
    new_population_set = mate_population(progenitor_list,
max_weight, objects_dict)

    mutated_pop = mutate_population(new_population_set,
max_weight,object_list, objects_dict)

0 140.0 364.18 09/12/24 15:31
100 32.0 309.62 09/12/24 15:31
200 21.0 299.47 09/12/24 15:31
300 54.0 320.89 09/12/24 15:31
400 32.0 304.21 09/12/24 15:31
500 78.0 320.9 09/12/24 15:31
600 12.0 318.54 09/12/24 15:31
700 57.0 321.24 09/12/24 15:31
800 64.0 316.02 09/12/24 15:31
900 74.0 318.47 09/12/24 15:31
1000 45.0 306.5 09/12/24 15:31
1100 54.0 302.42 09/12/24 15:31
1200 47.0 323.87 09/12/24 15:31
1300 76.0 300.41 09/12/24 15:31
1400 47.0 316.52 09/12/24 15:31
1500 54.0 333.84 09/12/24 15:31
1600 2.0 328.98 09/12/24 15:31
1700 32.0 304.49 09/12/24 15:31
1800 50.0 318.15 09/12/24 15:31
1900 64.0 317.2 09/12/24 15:31
2000 14.0 308.91 09/12/24 15:31
2100 32.0 324.82 09/12/24 15:31
2200 43.0 319.16 09/12/24 15:31
2300 57.0 316.62 09/12/24 15:31
```

2400	12.0	314.37	09/12/24	15:31
2500	12.0	303.14	09/12/24	15:31
2600	66.0	320.28	09/12/24	15:31
2700	19.0	302.17	09/12/24	15:31
2800	64.0	319.7	09/12/24	15:31
2900	74.0	321.43	09/12/24	15:31
3000	106.0	332.47	09/12/24	15:31
3100	64.0	309.67	09/12/24	15:31
3200	2.0	316.38	09/12/24	15:31
3300	50.0	306.35	09/12/24	15:31
3400	14.0	306.84	09/12/24	15:31
3500	12.0	306.26	09/12/24	15:31
3600	43.0	317.07	09/12/24	15:31
3700	12.0	313.58	09/12/24	15:31
3800	93.0	310.55	09/12/24	15:31
3900	32.0	269.12	09/12/24	15:31
4000	12.0	316.9	09/12/24	15:31
4100	96.0	332.95	09/12/24	15:31
4200	31.0	299.75	09/12/24	15:31
4300	31.0	323.25	09/12/24	15:31
4400	50.0	301.57	09/12/24	15:31
4500	57.0	307.58	09/12/24	15:31
4600	75.0	322.59	09/12/24	15:31
4700	87.0	313.54	09/12/24	15:32
4800	50.0	312.76	09/12/24	15:32
4900	69.0	328.56	09/12/24	15:32
5000	2.0	295.46	09/12/24	15:32
5100	12.0	307.89	09/12/24	15:32
5200	12.0	305.99	09/12/24	15:32
5300	32.0	319.15	09/12/24	15:32
5400	81.0	346.47	09/12/24	15:32
5500	66.0	316.0	09/12/24	15:32
5600	12.0	278.41	09/12/24	15:32
5700	12.0	308.65	09/12/24	15:32
5800	72.0	324.09	09/12/24	15:32
5900	52.0	313.44	09/12/24	15:32
6000	44.0	319.11	09/12/24	15:32
6100	62.0	295.17	09/12/24	15:32
6200	12.0	324.12	09/12/24	15:32
6300	43.0	306.21	09/12/24	15:32
6400	64.0	305.73	09/12/24	15:32
6500	51.0	311.73	09/12/24	15:32
6600	12.0	310.9	09/12/24	15:32
6700	64.0	315.81	09/12/24	15:32
6800	19.0	317.31	09/12/24	15:32
6900	50.0	315.68	09/12/24	15:32
7000	64.0	324.81	09/12/24	15:32
7100	12.0	314.03	09/12/24	15:32
7200	87.0	329.99	09/12/24	15:32
7300	14.0	321.7	09/12/24	15:32
7400	2.0	314.11	09/12/24	15:32
7500	74.0	323.88	09/12/24	15:32
7600	31.0	311.43	09/12/24	15:32

```

7700 32.0 325.35 09/12/24 15:32
7800 19.0 290.97 09/12/24 15:32
7900 2.0 317.41 09/12/24 15:32
8000 90.0 317.03 09/12/24 15:32
8100 14.0 320.51 09/12/24 15:32
8200 19.0 313.08 09/12/24 15:32
8300 34.0 322.63 09/12/24 15:32
8400 50.0 317.8 09/12/24 15:32
8500 12.0 290.06 09/12/24 15:32
8600 19.0 317.53 09/12/24 15:32
8700 57.0 328.55 09/12/24 15:32
8800 64.0 317.86 09/12/24 15:32
8900 68.0 327.04 09/12/24 15:32
9000 101.0 327.04 09/12/24 15:32
9100 2.0 314.04 09/12/24 15:33
9200 49.0 317.37 09/12/24 15:33
9300 95.0 314.45 09/12/24 15:33
9400 54.0 312.84 09/12/24 15:33
9500 32.0 305.0 09/12/24 15:33
9600 87.0 322.74 09/12/24 15:33
9700 54.0 311.62 09/12/24 15:33
9800 51.0 313.93 09/12/24 15:33
9900 12.0 291.94 09/12/24 15:33

```

Winner Genome 🏆

```

get_current_weight_value(best_solution[2][0], objects_dict)

(3.0, 723)

```

We'll pack into our backpack []:

```

best_solution[2][0]
#s_print(best_solution[2][0])

['Fork',
 'Glasses',
 'Blanket',
 'Coat',
 'Laptop',
 'Sleeping Bag',
 'Towel',
 'Shirts',
 'Knife',
 'Book',
 'Boots']

```

##5. Advantages and Applications of Genetic Algorithms

Genetic algorithms are powerful optimization and search techniques inspired by the principles of natural selection and genetics. They have found wide-ranging applications in various fields due to their ability to efficiently solve complex problems and explore solution spaces. In this text, we will explore some of the key advantages of genetic algorithms and their diverse applications.

Advantages of Genetic Algorithms:

1. **Global Optimization:** Genetic algorithms are particularly well-suited for global optimization problems, where the goal is to find the best solution among a vast number of possibilities. They excel at exploring large solution spaces, making them ideal for complex, multi-dimensional optimization tasks.
2. **Adaptability:** Genetic algorithms adapt and evolve over successive generations. This adaptability allows them to handle dynamic or changing problem environments effectively. They can continually refine their solutions as new information becomes available.
3. **Parallel Processing:** Genetic algorithms can be easily parallelized, allowing for efficient use of modern computing resources. This parallelism enables them to process a large number of potential solutions simultaneously, speeding up the optimization process.
4. **Exploration of Solution Space:** Genetic algorithms use techniques like mutation and crossover to explore different regions of the solution space. This exploration helps prevent them from getting stuck in local optima and increases the likelihood of finding the global optimum.

Applications of Genetic Algorithms:

1. **Machine Learning:** Genetic algorithms are used in machine learning for feature selection, hyperparameter tuning, and evolving neural network architectures. They aid in automating the process of optimizing complex models.
2. **Engineering and Design:** Genetic algorithms are employed in engineering design problems, such as aircraft design, where the goal is to find the best combination of design parameters that meet various constraints and objectives.
3. **Economics and Finance:** Genetic algorithms are applied in portfolio optimization, where they help in selecting a portfolio of financial assets to maximize returns while managing risk. They are also used in market forecasting.
4. **Bioinformatics:** Genetic algorithms are used for sequence alignment, protein folding, and optimizing genetic algorithms themselves. They are instrumental in solving complex biological problems.
5. **Game Playing:** In the field of artificial intelligence, genetic algorithms have been used to train agents in games. They can evolve strategies and behaviors that outperform human-designed approaches.
6. **Manufacturing and Scheduling:** Genetic algorithms are employed in optimizing production schedules, resource allocation, and supply chain management. They help companies streamline their operations and reduce costs.
7. **Robotics:** Genetic algorithms play a role in robot path planning, kinematic optimization, and control. They assist in designing robots capable of navigating complex environments efficiently.

In conclusion, genetic algorithms offer a versatile and powerful approach to solving complex optimization and search problems. Their adaptability, ability to explore large solution spaces, and applicability across diverse domains make them valuable tools in fields ranging from engineering and economics to biology and artificial intelligence. As computing power continues to advance, genetic algorithms are likely to find even more innovative applications in the future.

6. Real-world Applications of GA

Real-world Applications: Bridging Innovation and Practicality

Innovation often begins in the laboratory, but its true impact is felt when it reaches the real world, solving practical problems and improving people's lives. In this text, we'll explore a range of real-world applications across different fields, highlighting how these innovations have made a tangible difference.

1. Healthcare and Medicine:

The healthcare sector has witnessed groundbreaking advancements that directly impact patient care. Applications of artificial intelligence (AI) in medical diagnostics, including image analysis, have improved the accuracy and speed of disease detection. Personalized medicine, made possible by genomic sequencing, tailors treatment plans to individual patients, optimizing outcomes. Telemedicine has revolutionized healthcare access, particularly in remote areas, by enabling remote consultations and monitoring.

2. Transportation and Urban Planning:

Efficient transportation systems are essential for modern cities. Ride-sharing services and navigation apps have not only made commuting more convenient but have also reduced traffic congestion and emissions. Smart city initiatives incorporate IoT (Internet of Things) technologies for traffic management, waste reduction, and energy efficiency, resulting in cleaner and more sustainable urban environments.

3. Renewable Energy:

The transition to renewable energy sources, such as solar and wind power, has gained momentum in recent years. Solar panels on residential rooftops and large-scale wind farms contribute to a more sustainable energy grid. Energy storage solutions, like lithium-ion batteries, are crucial for storing excess energy generated during peak production times for later use, enhancing the reliability of renewable sources.

4. Education and E-Learning:

Technological advancements have reshaped education. E-learning platforms, online courses, and virtual classrooms provide flexible and accessible learning opportunities. These technologies have been particularly valuable during the COVID-19 pandemic, allowing for continued education in remote settings. Adaptive learning systems use AI to personalize instruction, tailoring content to individual students' needs and progress.

5. Environmental Conservation:

Environmental monitoring and conservation efforts benefit from cutting-edge technology. Drones and satellite imagery assist in tracking deforestation, wildlife populations, and changes in ecosystems. AI-powered algorithms analyze vast datasets to identify illegal fishing activity and poaching, aiding in the protection of endangered species.

6. Agriculture and Food Production:

Precision agriculture employs sensors, drones, and data analytics to optimize crop management. Farmers can monitor soil conditions, pest infestations, and crop health in real time, resulting in higher yields and reduced resource use. Genetic engineering and biotechnology have led to the development of drought-resistant crops and innovations in food production.

7. Finance and Fintech:

The financial industry has seen significant disruption through fintech innovations. Mobile payment apps, digital wallets, and blockchain technology have transformed the way we manage money and conduct transactions. Robo-advisors use AI to provide investment advice and manage portfolios efficiently.

These real-world applications represent just a fraction of the impact that innovation and technology have on our daily lives. As technology continues to advance, we can anticipate even more transformative changes, further enhancing our ability to address global challenges and improve the quality of life for people around the world. Innovations born in labs and research institutions continue to bridge the gap between theory and practice, making the future a promising one.

7. Summary

Summary: Unleashing the Potential of Genetic Algorithms in AI

Genetic algorithms, a powerful subfield of artificial intelligence, have proven to be versatile problem-solving tools with applications spanning various domains. In this exploration of genetic algorithms in AI, we've delved into their fundamental principles, advantages, and real-world applications, showcasing their ability to tackle complex optimization challenges.

At the core of genetic algorithms lies their emulation of the principles of natural selection and evolution. By representing potential solutions as genetic structures and iteratively evolving them through generations, genetic algorithms efficiently navigate vast solution spaces, searching for optimal or near-optimal solutions.

The advantages of genetic algorithms are numerous and compelling. They excel at global optimization, adapt to dynamic environments, harness parallel processing capabilities, and robustly explore solution spaces, mitigating the risk of getting trapped in local optima.

In the real world, genetic algorithms have demonstrated their prowess across diverse fields. They contribute to machine learning, enhancing model performance and automating complex tasks. In engineering and design, they optimize solutions in aeronautics, structural design, and more. In economics and finance, they guide investment strategies and market analysis. In bioinformatics, they assist in sequence analysis and drug discovery. They play pivotal roles in game playing, manufacturing, robotics, and more, shaping industries and driving innovation.

The future of genetic algorithms in AI holds great promise. As computing power continues to advance, genetic algorithms will tackle even larger and more complex optimization problems. Their synergy with emerging technologies like quantum computing may open new frontiers in AI research and applications.

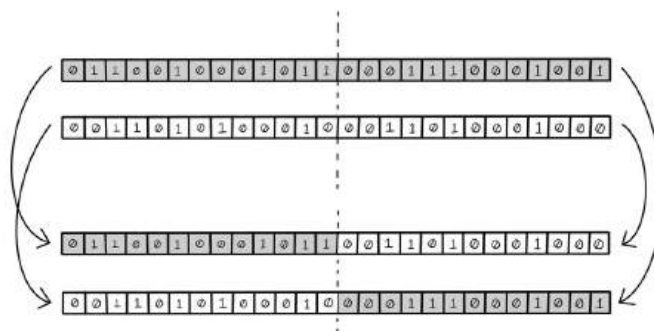
GAs use smart randomness to find good solutions fast.

Encoding is critical to the algorithm.

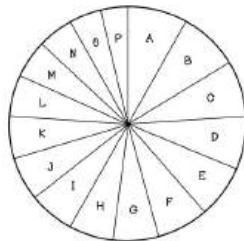
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
0	1	1	0	0	1	0	0	0	1	0	1	1	0	0	0	1	1	1	0	0	0	1	0	0	1

The fitness function is paramount to finding good solutions to the problem at hand.

Crossover aims to reproduce better solutions at each generation.

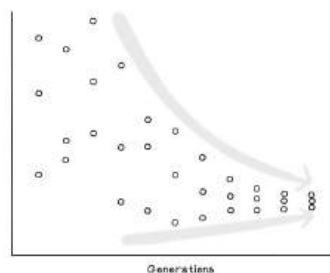


Selection favors stronger individuals but gives weaker ones an opportunity to possibly reproduce good solutions in future.



Roulette-wheel selection

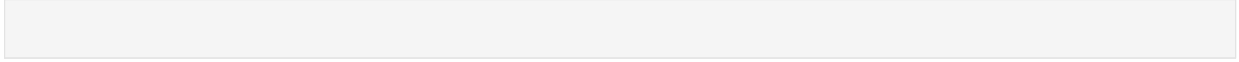
Explore at the start and exploit toward the end.



In conclusion, genetic algorithms represent a remarkable fusion of nature-inspired principles and computational ingenuity. Their capacity to solve intricate problems and discover optimal solutions across diverse domains cements their significance in the ever-evolving landscape

of artificial intelligence. As researchers and practitioners continue to harness their potential, genetic algorithms will undoubtedly play a pivotal role in shaping the AI-driven future, where complex problems are met with ingenious solutions, improving our world in ways we are only beginning to imagine.

0 - 0000 1 - 0001 2 - 0010 3 - 0011 4 - 0111 5 - 1111 6 -



Genetic Algorithm. Practical Tasks

Task: Solving the Traveling Salesman Problem Using Genetic Algorithm

Description: Write a Python program that uses a genetic algorithm to solve the Traveling Salesman Problem (TSP) for a small set of cities. The goal is to find the shortest route that visits each city exactly once and returns to the starting city.

Steps:

1. Define a set of cities and their coordinates.
2. Generate an initial population of random routes (order of cities).
3. Implement a fitness function to calculate the total distance of a route.
4. Perform selection, crossover, and mutation operations to create a new generation of routes.
5. Repeat the process for a certain number of generations.
6. Identify the best route in the final generation as the solution.

Optional Extensions:

- Experiment with different population sizes, mutation rates, and selection methods.

```
import random
import math
import matplotlib.pyplot as plt

def distance(city1, city2):
    """
    Calculates the Euclidean distance between two cities.

    Args:
        city1 (tuple): Coordinates of the first city (x, y).
        city2 (tuple): Coordinates of the second city (x, y).

    Returns:
        float: Euclidean distance between the two cities.
    """

    return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)

def total_distance(route, cities):
    """
    Calculates the total distance of a route that visits cities in
    order.

    Args:
        route (list): List of city indices in the route.
        cities (list): List of city coordinates.

    Returns:
        float: Total distance of the route.
    """
```

```

    """

    return total

def generate_initial_population(num_routes, num_cities):
    """
    Generates an initial population of random routes.

    Args:
        num_routes (int): Number of routes in the population.
        num_cities (int): Number of cities in the route.

    Returns:
        list: List of random routes (lists of city indices).
    """

    return initial_population

def selection(population, cities):
    """
    Selects individuals from the population for reproduction using
    tournament selection.

    Args:
        population (list): List of routes (lists of city indices).
        cities (list): List of city coordinates.

    Returns:
        list: List of selected routes for reproduction.
    """

    return selected_routes

def crossover(parent1, parent2):
    """
    Performs one-point crossover between two parent routes to create
    two child routes.

    Args:
        parent1 (list): List of city indices in the first parent
        route.
        parent2 (list): List of city indices in the second parent
        route.

```

```

    Returns:
        child1 (list): List of city indices in the first child
route.
        child2 (list): List of city indices in the second child
route.
    """

    return child1, child2

def mutation(route):
    """
    Performs swap mutation on a route by swapping two randomly
selected cities.

    Args:
        route (list): List of city indices in the route.

    Returns:
        mutated_route (list): List of city indices in the mutated
route.
    """

    return mutated_route

# Define your list of city coordinates
cities = [(0, 0), (2, 4), (5, 8), (9, 1), (3, 6)]

# Set parameters
num_routes = 50
num_generations = 100
mutation_rate = 0.1

# Generate initial population
population = generate_initial_population(num_routes, len(cities))

# Run genetic algorithm for a certain number of generations
for generation in range(num_generations):
    # Selection
    selected_routes = selection(population, cities)

    # Crossover
    new_generation = []
    for i in range(0, len(selected_routes), 2):
        if i + 1 < len(selected_routes):
            child1, child2 = crossover(selected_routes[i],
selected_routes[i + 1])
            new_generation.append(child1)

```

```

        new_generation.append(child2)

    # Mutation
    mutated_generation = []
    for route in new_generation:
        if random.random() < mutation_rate:
            mutated_route = mutation(route)
            mutated_generation.append(mutated_route)
        else:
            mutated_generation.append(route)

    # Update the population with the new generation
    population = mutated_generation

# Find the best route in the final generation
    best_route = min(population, key=lambda route: total_distance(route,
    cities))

# Print the best route and its total distance
    print(f"Best Route: {best_route}")
    print(f"Total Distance: {total_distance(best_route, cities)}")

```

References

1. **Russell S. J., Norvig P.** *Artificial Intelligence: A Modern Approach*. 4th ed. Upper Saddle River : Pearson Education, 2020. 1136 p.
2. **Poole D. L., Mackworth A. K.** *Artificial Intelligence: Foundations of Computational Agents*. 3rd ed. Cambridge : Cambridge University Press, 2023. 898 p.
3. **Chowdhary K. R.** *Fundamentals of Artificial Intelligence*. New Delhi : Springer, 2020. 716 p.
4. **Gupta I., Nagpal G.** *Artificial Intelligence and Expert Systems: Techniques and Applications for Problem Solving*. Dulles, VA : Mercury Learning and Information, 2020. 424 p.
5. **Murphy K. P.** *Probabilistic Machine Learning: An Introduction*. Cambridge, MA : MIT Press, 2022. 864 p.
6. **Du K.-L., Swamy M. N. S.** *Neural Networks and Statistical Learning*. 2nd ed. London : Springer, 2019. 988 p.
7. **Drori I.** *The Science of Deep Learning*. Cambridge : Cambridge University Press, 2023. 360 p.
8. **Plaat A.** *Deep Reinforcement Learning*. Singapore : Springer, 2022. 406 p.
9. **Kejriwal M., Knoblock C. A., Szekely P.** *Knowledge Graphs: Fundamentals, Techniques, and Applications*. Cambridge, MA : MIT Press, 2021. 568 p.
10. **Kamath U., Liu J.** *Explainable Artificial Intelligence: An Introduction to Interpretable Machine Learning*. Cham : Springer, 2021. 310 p.