Table of contents

Table of contents 1
Semester 1 Handbook 7
Module A 7
1. Datatypes in Python: 7
    1.1 Integer (int) 7
    1.2 Float (float) 7
    1. 3 String (str) 7
    1. 4 Boolean (bool) 8
    1.5 List (list) 8
    1.6 Tuple 8
    1.7 Dictionary (dict) 8
    1. 8 Set (set) 9
    1.9 None Type (None) 9
    1.10 Complex (complex) 9
2. Logical Operators Overview 10
    2.1 and 10
    2.2 or 10
    2.3 not 10
3. Comparison Operators 10
4. Arithmetic Operators in Python 10
    4.1 Basic Arithmetic 11
    4.2 Advanced Operations 11
    4.3 Operations on Python Data Types 11
        4.3.1 Strings 11
        4.3.2 Lists 12
        4.3.3 Tuples 12
        4.3.4 Sets 12
5. Syntax of if-elif-else Statements 13
6. For Loop 13
7. While Loop 13
8. break Statement: 13
9. continue Statement: 14
10. pass Statement and Error Handling: 14
    10.1 pass Statement 14
    10.2 Error Handling (try…except…finally) 14
11. General Functions for Data Types 15
12. List Comprehensions 15
13. Dictionary Comprehension : 15

14. Lambda Functions:.....15
sorted() Function:.....16
15. Recursion :.....16
15.1 What is Recursion?.....16
15.2 How does Recursion work?.....16
15.3 Example: Fibonacci Sequence using Recursion.....16
16. Variable-Length Arguments (*args) and Keyword Arguments (**kwargs):.....16
16.1 Keyword Arguments (**kwargs):.....17
16.2 Combining *args and **kwargs.....17
16.3 Using with Functions:.....17
17. File Handling in Python.....17
17.1 Opening and Closing Files:.....17
17.2 Modes:.....17
17.3 Reading Files:.....17
17.4 Writing Files:.....17
17.5 Context Manager:.....18
17.6 Opening a File and Reading Content:.....18
17.7 Writing to a File:.....18
17.8 Appending to a File:.....18
17.9 Reading File Line by Line:.....18
17.10 Seek and Tell:.....18
18. Sorting Techniques and Analysis:.....19
18.1 Bubble Sort Concept.....19
18.2 Quick Sort Concept.....19
18.3 Merge Sort Concept.....20
18.4 Insertion Sort Concept:.....21
18.5 Selection Sort.....22
19. Markov Chain:.....22
19.1 States:.....23
19.2 Transition Probabilities:.....23
19.3 Initial State Distribution:.....23
19.4 Stationary Distribution:.....23
Final Answer.....24
20. NumPy.....25
20.1 Array.....25
20.2 Difference between NumPy arrays and lists.....25
20.3 Higher dimension arrays.....25
20.4 Array Properties/Attributes.....25
20.5 Accessing Array Elements.....26
20.6 Conditional Indexing.....26
20.7 Mutability.....26

20.8 Referencing....26
20.9 Reshaping....26
20.10 Data Types....27
Initialization....27
20.11 Zeros and Ones....27
20.12 Range....27
20.13 Operations....27
20.14 Matrix....28
20.15 Transpose....28
20.16 Determinant....28
20.17 Universal Functions....28
20.18 Understanding the axis Parameter....28
21. Random Module....29
21.1 Distribution....29
21.2 Seed....29
22. Statistical Analysis....29
22.1 Mean....29
22.2 Median and Mode....30
22.3 Standard deviation and Variance....30
22.4 Statistical Range....30
22.5 Visualization with Matplotlib & Seaborn....30
22.6 Introduction to Seaborn :....31
22.7 Visualizing Algorithm Efficiency (Required Addition)....31
22.8 Basic Probability Concepts (Required Addition)....31
22.9 Hypothesis Testing....31
Common Formulas....32
23. Pandas DataFrame....33
23.1 Accessing Multiple Columns Concept....33
23.2 Introduction to Pandas DataFrames....33
23.3 Creating a DataFrame:....33
23.4 Basic Information About the DataFrame:....33
23.5 Accessing Data....33
23.6 Filtering Data:....34
23.7 Slicing Rows....34
23.8 Accessing First Rows....35
23.9 Accessing Last Rows....35
23.10 Statistical Analysis on Data....35
Syntax....36
Examples....36
23.12 Reading Data from CSV....36
23.13 Getting DataFrame Information....37

23.14 Finding Specific Values in DataFrame..... 37
23.14 Data Aggregation with .groupby()..... 37
24. Object-Oriented Programming (OOP)..... 37
24.1 Classes and Objects..... 37
24.2 Encapsulation..... 38
24.3 Inheritance..... 39
24.4 Polymorphism..... 40
24.5 Abstraction..... 40
24.6 Method Overloading vs. Overriding..... 41
24.7 Access Modifiers..... 42
24.8 Summary Table:..... 43
Module B..... 44
1. Regression: Linear and Polynomial..... 44
1.1 Linear Regression..... 44
1.2 Polynomial Regression..... 44
1.3 Overfitting vs. Underfitting..... 44
1.4 Logistic Regression..... 44
1.5 L1 & L2 Regularization..... 45
2. Principal Component Analysis (PCA) & t-SNE..... 45
2.1 How PCA Works: A Step-by-Step Guide..... 46
2.1.1 Step 1: Standardize the Data..... 46
2.1.2 Step 2: Compute the Covariance Matrix..... 46
2.1.3 Step 3: Find Eigenvectors and Eigenvalues..... 46
2.1.4 Step 4: Select Principal Components..... 47
2.1.5 Step 5: Transform the Data..... 47
2.1.6 Summary & Key Properties..... 47
2.2 t-SNE (t-Distributed Stochastic Neighbor Embedding)..... 47
3. PageRank..... 47
3.1 Graph Theory Basics (for PageRank)..... 47
3.2 Process..... 48
3.3 Components and Their Effect..... 48
3.3.1 The PageRank Formula..... 48
3.4 Worked Numerical Example (with Damping Factor)..... 49
Iteration 1:..... 49
4. Neural Networks..... 50
4.1 Foundations of the Perceptron..... 50
4.2 Enhancing the Perceptron with Learning Algorithms..... 50
4.3 Neuron: Basic Building Block..... 50
4.4 Components of a Neuron..... 51
4.5 Mathematical Representation..... 52
4.6 Structure of a Neural Network..... 52

5. Forward and Backward Propagation....53
5.1 Overview....53
5.1.1 Gradient Descent Variants....53
5.2 Forward Pass....54
5.3 Backward Pass....55
5.4 Error Calculation....56
5.5 Activation Functions....56
5.5.1 Sigmoid:....56
Tanh:....56
ReLU:....56
Leaky ReLU:....56
ELU:....56
Softmax (Required Addition):....57
6. Convolutional Neural Networks (CNN)....57
6.1 Conceptual Advantages....57
6.2 The Convolution Operation....57
6.3 Pooling Layers....57
6.4 Output Size Calculation....57
6.5 Common Architectures....58
6.6 Keras/Deep Learning Library Syntax....58
7. Support Vector Machine (SVM)....58
7.1 Process....58
7.2 Formula....59
7.3 Important Properties....59
7.4 The Kernel Trick....59
8. Supervised Learning - Evaluation Metrics....59
8.1 Confusion Matrix....59
8.2 Accuracy....60
8.3 Precision....60
8.4 Recall (Sensitivity)....60
8.5 F1 Score....60
8.6 Mean Absolute Error (MAE) (For Regression)....60
8.7 Mean Squared Error (MSE) (For Regression)....61
9. Evaluation Metrics for Unsupervised Learning....61
9.1 Silhouette Score....61
9.2 Reconstruction Error (Dimensionality Reduction)....62
9.3 Important Properties....62
10. K-Means Clustering....62
10.1 Process....62
10.2 Formula....63
10.3 Important Properties....63

10.4 The Elbow Method..... 64
11. Hierarchical Clustering.....64
11.1 Agglomerative (Bottom-Up) Process..... 64
11.2 Linkage Methods..... 64
12. Density-Based Spatial Clustering of Applications with Noise (DBSCAN)..... 64
12.1 Key Concepts.....64
12.2 Process..... 65
13. Computer Vision..... 65
13.1 Key Tasks.....65
13.2 Common Architectures.....65
13.2.1 LeNet-5..... 65
13.2.2 AlexNet..... 65
13.2.3 VGG16..... 65
13.2.4 GoogLeNet (Inception)..... 65
13.2.5 Residual Networks (ResNet).....65
13.2.6 Densely Connected Convolutional Networks (DenseNet).....66

Semester 1 Handbook

Module A

1. Datatypes in Python:

1.1 Integer (int)

- Description: Integers represent whole numbers, both positive and negative, without decimals.
- Example:

```
age = 25
temperature = -5
```

- Explanation: In this example, age and temperature are integers because they don't have any decimal parts.

1.2 Float (float)

- Description: Floats represent real numbers that have decimal points.
- Example:

```
price = 9.99
weight = 70.5
```

- Explanation: Both price and weight have decimal parts, which makes them float values.

1. 3 String (str)

- Description: Strings are sequences of characters enclosed within quotes. They are used for text.
- Example:

```
name = "Alice"
greeting = "Hello, world!"
```

- Explanation: Here, name and greeting are strings because they contain characters enclosed in quotes.

1. 4 Boolean (bool)

- Description: Booleans represent one of two values: True or False. They are often used in conditional statements,
- Example:

```
is_active = True
has_permission = False
```

- Explanation: `is_active` and `has_permission` are Boolean values, representing `True` or `False`.

1.5 List (list)

- Description: Lists are ordered collections of items, which can be of any data type. Lists are mutable, meaning their contents can be changed.
- Example:

```
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
```

- Explanation: Here, fruits is a list of strings, and numbers is a list of integers. Lists are written within square brackets `[]`

1.6 Tuple

- Description: Tuples are ordered collections of items, similar to lists, but they are immutable, meaning they cannot be changed once created.
- Example:

```
coordinates = (10.5, 20.8)
colors = ("red", "green", "blue")
```

- Explanation: coordinates and colors are tuples. They are enclosed within parentheses () and cannot be modified.

1.7 Dictionary (dict)

- Description: Dictionaries are collections of key-value pairs. They are unordered and accessed by unique keys rather than by index.
- Example:

```
person = {"name": "Alice", "age": 25, "city": "New York"}
```

- Explanation: person is a dictionary where each piece of data (like "name" and "age") is associated with a unique key. Dictionaries are written within curly braces {}.

1. 8 Set (set)

- Description: Sets are collections of unique items. They are unordered, meaning there's no guaranteed order for items. Sets automatically eliminate duplicate values.
- Example:

```
unique_numbers = {1, 2, 3, 4, 4, 5}
# This becomes {1, 2, 3, 4, 5}
```

- Explanation: unique_numbers is a set. Even though 4 is added twice, the set stores only unique values {1, 2, 3, 4, 5}.

1.9 None Type (None)

- Description: None represents the absence of a value or a null value. It is often used as a placeholder or to indicate "no value."
- Example:

```
result = None
```

- Explanation: result is set to None, which means it currently has no value.

1.10 Complex (complex)

- Description: Complex numbers consist of a real and an imaginary part, represented as $a + bj$ where a is the real part and bj is the imaginary part. **The unit 'j' represents the imaginary unit, defined as** $j = \sqrt{-1}$
- Example:
  $z = 3 + 4j$
- Explanation: z is a complex number where 3 is the real part and 4j is the imaginary part.
  Basic operations include
  Addition:
  $(a + bj) + (c + dj) = (a + c) + (b + d)j$

  Multiplication:
  $(a + bj) * (c + dj) = (ac - bd) + (ad + bc)j$

---

2. Logical Operators Overview

2.1 and

Returns `True` if both conditions are `True`.

```
if has_lantern and has_map:
    print("You are ready!")
```

2.2 or

Returns `True` if at least one condition is `True`.

```
if has_lantern or has_map:
    print("You can proceed carefully.")
```

2.3 not

Reverses the Boolean value.

```
if not has_key:
    print("You need the key.")
```

---

3. Comparison Operators

Operator	Description	Example	Output
<	Less than	5<10	True
>	Greater than	10>5	True
<=	Less than or equal	5<=5	True
>=	Greater than or equal	5>=10	False
==	Equal to	5==5	True
!=	Not equal to	5!=10	True

---

4. Arithmetic Operators in Python

Python provides several operators for performing arithmetic operations.

Operator	Description	Example	Result
+	Addition	5 + 3	8

Operator	Description	Example	Result
-	Subtraction	5 - 3	2
*	Multiplication	5 * 3	15
/	Division (float result)	5 / 3	1.666…
//	Floor Division	5 // 3	1
%	Modulus (Remainder)	5 % 3	2
**	Exponentiation	5 ** 3	125

Examples of Usage

4.1 Basic Arithmetic

```
a = 10
b = 3
print(a + b) # Output: 13
print(a - b) # Output: 7
```

4.2 Advanced Operations

```
x = 7
y = 2
print(x**y) # 7 to the power of 2 = 49
print(x // y) # Floor division = 3
print(x % y) # Remainder = 1
```

4.3 Operations on Python Data Types

Python provides several operations for different data types. Here's a quick reference:

4.3.1 Strings

Operation	Description	Example	Result
Indexing	Access a specific character	"hello"[1]	'e'
Slicing	Extract substring	"hello"[1:4]	'ell'
Concatenation	Combine strings	"hello" + "world"	'hello world'
Repetition	Repeat string	"ha" * 3	'hahaha'

4.3.2 Lists

Operation	Description	Example	Result
Indexing	Access specific element	[1, 2, 3][0]	1
Slicing	Extract sublist	[1, 2, 3, 4][1:3]	[2, 3]
Append	Add element to end	lst=[1]; lst.append(2)	[1, 2]
Extend	Add multiple elements	lst.extend([3, 4])	[1, 2, 3, 4]

4.3.3 Tuples

Operation	Description	Example	Result
Indexing	Access specific element	(1, 2, 3)[1]	2
Slicing	Extract sub-tuple	(1, 2, 3, 4)[1:3]	(2, 3)
Concatenation	Combine tuples	(1, 2) + (3, 4)	(1, 2, 3, 4)
Repetition	Repeat tuple	(1, 2) * 2	(1, 2, 1, 2)

4.3.4 Sets

Operation	Description	Example	Result
Add	Add element	s={1,2}; s.add(5)	{1, 2, 5}
Remove	Remove element	s={1,2,3}; s.remove(3)	{1, 2}
Union	Combine sets	{1, 2} | {3, 4}	{1, 2, 3, 4}
Intersection	Common elements	{1, 2} & {2, 3}	{2}

4.9 Dictionaries

Operation	Description	Example	Result
Access Value	Get value for a key	d["key"]	‘value’
Add/Update Pair	Add or update key-value pair	d["new_key"] = "new_value"	{'key': 'value', 'new_key': 'new_value'}
Keys/Values	Get all keys or values	d.keys() / d.values()	['key'] / ['value']
Delete Pair	Remove a key-value pair	del d["key"]	{}

5. Syntax of if-elif-else Statements

```
if condition1:
    # Code block for condition1elif condition2:
    # Code block for condition2else:
    # Code block for when none of the above conditions are true
```

---

6. For Loop

Used to iterate over sequences like lists, tuples, strings, or ranges.

Syntax:

```
for variable in sequence:
    # Code to execute
```

---

7. While Loop

Used to repeatedly execute a block of code as long as a condition is True.

Syntax:

```
while condition:
    # Code to execute
```

---

8. break Statement:

The break statement is used to exit the loop early, regardless of the loop's condition. It terminates the loop immediately and control is transferred to the next statement after the loop.

Syntax:

```
for i in range(10):
    if i == 5:
        break # exit the loop when i is 5
```

---

9. continue Statement:

The continue statement skips the current iteration of the loop and continues to the next iteration, without executing the remaining code for the current loop iteration.

Syntax:

```
for i in range(10):
    if i == 5:
        continue # skip the rest of the loop when i is 5
    print(i)
```

---

10. pass Statement and Error Handling:

10.1 pass Statement

The pass statement is a placeholder. It is used when no action is required and is usually found in places where code is syntactically required but no action is needed.

Syntax:

```
for i in range(10):
    if i == 5:
        pass # do nothing
    else:
        print(i)
```

10.2 Error Handling (try…except…finally)

Python uses a try...except block to handle errors (exceptions) gracefully when they occur.

- try: This block contains the code that might raise an exception.
- except: If an exception occurs in the try block, the code in the except block is executed, preventing the program from crashing.
- finally: This block is optional and will execute regardless of whether an exception was raised. It is typically used for cleanup operations like closing files.

Example:

```
try:
    # This code might cause an error    result = 10 / 0
except ZeroDivisionError:
    # This code runs only if a ZeroDivisionError occurs
    print("You cannot divide by zero!")
finally:
    # This code always runs
    print("This will always execute, for cleanup.")
```

---

11. General Functions for Data Types

Common Functions

- `len()`: Returns the number of items in an object.
- `max()`: Returns the largest item in an object.
- `min()`: Returns the smallest item in an object.
- `sum()`: Returns the sum of items (only for numeric elements).
- `sorted()`: Returns a sorted list of elements from an object.

---

12. List Comprehensions

List comprehensions provide a concise way to create lists. They combine loops and conditional statements into a single line of code. **Syntax Example**

```
numbers = [1, 2, 3, 4, 5]
squares = [x**2 for x in numbers if x > 2]
print(squares) # Output: [9, 16, 25]
```

- **Explanation:** The above code iterates through the list `numbers`, selects elements greater than 2, squares them, and stores the results in the new list `squares`.

---

13. Dictionary Comprehension :

A concise way to create dictionaries using loops or conditions within a single line.

- **Syntax:** `{key: value for (key, value) in iterable if condition}`
- **Example:**

```
numbers = [1, 2, 3, 4, 5]
squares = {num: num**2 for num in numbers if num > 2}
print(squares)
```

- **Output:** `{3:9, 4: 16, 5:25}`
- **Explanation:** This dictionary comprehension iterates through the list `numbers` and includes only those numbers greater than 2. For each number, it creates a key-value pair where the key is the number (`num`) and the value is the square of the number `num**2`.

---

14. Lambda Functions:

- **Purpose:** A small, anonymous function defined with the `lambda` keyword.

- **Syntax:** lambda arguments: expression
- Commonly used in short operations, especially with sorted() and map().
- **Example:**

```
square = lambda x: x**2
print(square(5)) # Output: 25
```

sorted() Function:

- **Purpose:** Sorts elements in an iterable (e.g., a list) based on a key.
- **Syntax:** sorted(iterable, key=function, reverse=False)

---

15. Recursion :

15.1 What is Recursion?

Recursion is a programming technique in which a function calls itself in order to solve smaller instances of the same problem. A recursive function typically has two parts:

1. **Base Case:** The condition under which the function stops calling itself and returns a result. Without a base case, the function would call itself indefinitely.
2. **Recursive Case:** The part where the function calls itself with modified arguments, progressively solving smaller parts of the problem.

15.2 How does Recursion work?

- A recursive function reduces the problem into a simpler or smaller version of the same problem.
- The function keeps calling itself with simpler inputs until it hits the base case.
- The function then starts returning the results and building up the solution step by step.

15.3 Example: Fibonacci Sequence using Recursion

The Fibonacci sequence is a series of numbers in which each number is the sum of the two preceding ones. The sequence starts with 0 and 1: 0, 1, 1, 2, 3, 5, 8, … The recursive formula for Fibonacci is:

- $fib(n) = fib(n-1) + fib(n-2)\ for\ n > 1$
- Base cases: $fib(0) = 0,\ fib(1) = 1$

---

16. Variable-Length Arguments (*args) and Keyword Arguments (**kwargs):

- Allows a function to accept any number of positional arguments.

- **Syntax:** `*args` collects extra arguments into a tuple.

16.1 Keyword Arguments (**kwargs):

- Allows a function to accept any number of keyword arguments.
- **Syntax:** `**kwargs` collects extra named arguments into a dictionary.

16.2 Combining *args and **kwargs

- You can use both `*args` and `**kwargs` in the same function to handle both positional and keyword arguments.

```
def describe(*args, **kwargs):
    print("Positional arguments:", args)
    print("Keyword arguments:", kwargs)

describe("artifact1", "artifact2", rarity="rare", year=1985)
# Output:
# Positional arguments: ('artifact1', 'artifact2')
# Keyword arguments: {'rarity': 'rare', 'year': 1985}
```

16.3 Using with Functions:

- `*args` and `**kwargs` make your functions flexible and reusable by allowing them to accept varying numbers and types of inputs.

---

17. File Handling in Python

Key Concepts

17.1 Opening and Closing Files:

Use `open()` to open a file and `close()` to close it.

17.2 Modes:

- `'r'`: Read (default mode).
- `'w'`: Write (overwrites file if it exists).
- `'a'`: Append (adds content to the end of the file).
- `'r+'`: Read and write.

17.3 Reading Files:

Use `read()`, `readline()`, or `readlines()` to read file content.

17.4 Writing Files:

Use `write()` to write data to a file.

17.5 Context Manager:

Use `with` to handle files automatically (no need to call `close()`).

Examples

17.6 Opening a File and Reading Content:

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content) # Prints the entire file content
```

17.7 Writing to a File:

```
with open("example.txt", "w") as file:
    file.write("Hello, World!")
```

17.8 Appending to a File:

```
with open("example.txt", "a") as file:
    file.write("\nThis is a new line.")
```

17.9 Reading File Line by Line:

```
with open("example.txt", "r") as file:
    for line in file:
        print(line.strip()) # Removes leading/trailing whitespace
```

17.10 Seek and Tell:

Manage the file pointer using `seek()` and find its position with `tell()`.

```
with open("example.txt", "r") as file:
    print(file.tell()) # Outputs the current file pointer position
    file.seek(5) # Moves the pointer to the 5th byte
    print(file.read()) # Reads from the 5th byte onwards
```

---

18. Sorting Techniques and Analysis:

18.1 Bubble Sort Concept

Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

- **Time Complexity**
  - **Best case:** $O(n)$ - This occurs when the list is already sorted. The algorithm only needs one pass through the list to confirm that no swaps are needed.
  - **Worst case:** $O(n^2)$ - This occurs when the list is sorted in reverse order, meaning the algorithm has to compare and swap each element in every pass.
- **Space Complexity**
  - $O(1)$ (in-place sorting).
- **How It Works**
  1. Compare each pair of adjacent elements.
  2. If the first element is larger than the second, swap them.
  3. Repeat the process for every element until no swaps are needed.
  4. The largest element "bubbles" up to the correct position after each pass.
- Example

  Initial list: [5, 2, 9, 1, 5, 6]
  - After 1st pass: [2, 5, 1, 5, 6, 9]
  - After 2nd pass: [2, 1, 5, 5, 6, 9]
  - After 3rd pass: [1, 2, 5, 5, 6, 9] (sorted)

18.2 Quick Sort Concept

Quick Sort is a divide-and-conquer algorithm. It picks a "pivot" element and partitions the list into two sublists: elements smaller than the pivot and elements greater than the pivot. Then, it recursively sorts the sublists.

- **Time Complexity**
  - **Best case:** $O(nlogn)$ - This occurs when the pivot divides the list into two nearly equal halves. This results in balanced partitioning at each step, leading to efficient sorting.
  - **Worst case:** $O(n^2)$ - This occurs when the pivot is the smallest or largest element in the list, causing the partitioning to be unbalanced… leading to poor performance.
- **Space Complexity**

- $O(logn)$ (in-place sorting).

- **How It Works**

  1. Choose a pivot element from the list (commonly the last element).
  2. Partition the list into two sublists: one with elements smaller than the pivot and one with elements greater than the pivot.
  3. Recursively apply the same process to the sublists.
  4. Once the sublists have been sorted, the entire list is sorted.

- Example

  Initial list: [10, 80, 30, 90, 40, 50, 70]

  - Pivot: 70
  - Partitioned: [10, 30, 40, 50, 70, 90, 80]
  - Now recursively sort the sublists: [10, 30, 40, 50] and [90, 80]
  - Final sorted list: [10, 30, 40, 50, 70, 80, 90]

18.3 Merge Sort Concept

Merge Sort is another divide-and-conquer algorithm. It divides the list into two halves, recursively sorts each half, and then merges the two sorted halves into a single sorted list.

- **Time Complexity**

  - **Best case:** $O(nlogn)$ - The time complexity is always $O(nlogn)$ because the list is always divided in half…
  - **Worst case:** $O(nlogn)$ - Merge Sort performs the same regardless of the initial order of elements…

- **Space Complexity**

  - $O(n)$ (requires extra space for merging).

- **How It Works**

  1. Divide the list into two halves.
  2. Recursively sort each half.
  3. Merge the sorted halves back together by comparing elements from both halves and adding them to the sorted list.

- Example

  Initial list: [38, 27, 43, 3, 9, 82, 10]

  - Split: [38, 27, 43], [3, 9, 82, 10]
  - Recursively sort: [27, 38, 43], [3, 9, 10, 82]
  - Merge: [3, 9, 10, 27, 38, 43, 82]

18.4 Insertion Sort Concept:

Insertion Sort is a simple, comparison-based sorting algorithm that builds the final sorted array one item at a time. It is analogous to the way people sort playing cards in their hands.

- **Time Complexity:**
  - **Best case:** $O(n)$ - Occurs when the list is already sorted. Only one comparison is needed per element.
  - **Worst case:** $O(n^2)$ - Occurs when the list is sorted in reverse order. Each element is compared with all previous elements.
- **Space Complexity:** $O(1)$ (in-place sorting).
- **How It Works:**
  1. Divide the list into a "sorted" and "unsorted" section. Initially, the first element is considered sorted.
  2. Take the first element from the unsorted section and insert it into its correct position in the sorted section.
  3. Repeat the process for all elements in the unsorted section until the entire list is sorted.
- Clarified Example:

  Initial list: [12, 11, 13, 5, 6]

  1. Sorted: [12] Unsorted: [11, 13, 5, 6] Take 11. Compare to 12 (smaller). Insert 11 before 12.

     Result: [11, 12, 13, 5, 6]

  2. Sorted: [11, 12] Unsorted: [13, 5, 6] Take 13. Compare to 12 (larger). Stays in place.

     Result: [11, 12, 13, 5, 6]

  3. Sorted: [11, 12, 13] Unsorted: [5, 6]. Take 5. Compare to 13, 12, 11. It's smaller than all. Insert 5 at the beginning.

     Result: [5, 11, 12, 13, 6]

  4. Sorted: [5, 11, 12, 13]Unsorted: [6] Take 6. Compare to 13, 12, 11 (smaller). Compare to 5 (larger). Insert 6 after 5.

     Result: [5, 6, 11, 12, 13]

  Final sorted list: [5, 6, 11, 12, 13]

18.5 Selection Sort

- Concept:

  Selection Sort is a comparison-based sorting algorithm that repeatedly selects the smallest (or largest) element from the unsorted section and moves it to the sorted section.

- **Time Complexity:**
  - **Best case:** $O(n^2)$ - Even in the best case, the algorithm performs the same number of comparisons as it does in the worst case.
  - **Worst case:** $O(n^2)$ Each element is compared to all others to find the minimum.
- **Space Complexity:** $O(1)$ (in-place sorting).
- **How It Works:**
  1. Divide the list into a "sorted" and "unsorted" section. Initially, the entire list is unsorted.
  2. Find the smallest element in the unsorted section and swap it with the first element of the unsorted section.
  3. Move the boundary between the sorted and unsorted sections one element to the right.
  4. Repeat the process until the entire list is sorted.
- Example:

  Initial list: [64, 25, 12, 22, 11]

  1. Find the smallest element (11) and swap it with 64. Result: [11, 25, 12, 22, 64]
  2. Find the next smallest element (12) and swap it with 25. Result: [11, 12, 25, 22, 64]
  3. Find the next smallest element (22) and swap it with 25. Result: [11, 12, 22, 25, 64]
  4. The remaining unsorted list is [25, 64]. 25 is the smallest. It is already in place.

  Final sorted list: [11, 12, 22, 25, 64]

---

19. Markov Chain:

A Markov chain is a stochastic process that satisfies the **Markov property**:

the future state depends **only** on the present state, not on the sequence of past events (it is *memoryless*).

Key Concepts

19.1 States:

The possible conditions the system can be in (e.g., {Sunny, Rainy}).

19.2 Transition Probabilities:

Probabilities of moving from one state to another, represented in a **Transition Matrix (P)**.

Entry P_{ij} = probability of moving from state i to state j.

Example:

$$P = \begin{pmatrix} P(S_1 \to S_1) & P(S_1 \to S_2) & P(S_1 \to S_3) \\ P(S_2 \to S_1) & P(S_2 \to S_2) & P(S_2 \to S_3) \\ P(S_3 \to S_1) & P(S_3 \to S_2) & P(S_3 \to S_3) \end{pmatrix}$$

19.3 Initial State Distribution:

A vector representing probabilities of starting in each state at $t = 0$:

$$\pi_0 = \left[P\left(S_1 \, at \, t = 0\right), P\left(S_2 \, at \, t = 0\right), P\left(S_3 \, at \, t = 0\right)\right]$$

19.4 Stationary Distribution:

A probability distribution vector π that does not change as the system evolves:

$$\pi P = \pi$$

Steps to Solve Markov Chain Problems (19.6)

1. **Identify States:** List all possible states.
2. **Define Transition Matrix (P):** Construct matrix of transition probabilities.
3. **Analyze Transitions:** Compute probabilities for future states:

   $\pi_n = \pi_0 \cdot P^n$

4. **Compute Stationary Distribution:** Solve

   $\pi P = \pi, \quad \sum \pi_i = 1$

Worked Example

Problem: A town's weather is either *Sunny (State 0)* or *Rainy (State 1)*.

- If Sunny → 90% chance it stays Sunny, 10% chance of Rain.
- If Rainy → 50% chance it stays Rainy, 50% chance of Sun.

Transition Matrix:

$$P = \begin{bmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{bmatrix}$$

Step 1: Initial State

Day 0 is Sunny →

$\pi_0 = [1.0, 0.0]$

Step 2: Day 1

$$\pi_1 = \pi_0 \times P = [1.0, 0.0] \times \begin{bmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{bmatrix} = [0.9, 0.1]$$

Step 3: Day 2

$$\pi_2 = \pi_1 \times P = [0.9, 0.1] \times \begin{bmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{bmatrix}$$

$$\pi_2 = [(0.9 \times 0.9 + 0.1 \times 0.5), (0.9 \times 0.1 + 0.1 \times 0.5)]$$

$$\pi_2 = [0.86, 0.14]$$

Final Answer

On **Day 2**, there is:

- **86% chance of Sun**
- **14% chance of Rain**

---

20. NumPy

NumPy (Numerical Python) is a foundational library for numerical and scientific computing in Python. It provides robust support for large multi-dimensional arrays and matrices, along with an extensive collection of high-level mathematical functions to manipulate these arrays. NumPy is highly optimized for performance and is a cornerstone in fields like data science, machine learning, and scientific research.

20.1 Array

Basic array creation. An array is a central data structure in NumPy, enabling efficient storage and manipulation of homogeneous data types.

```
import numpy as np
# Creating a 1D array
arr = np.array([1, 2, 3, 4])
# Creating a 2D array
arr_2d = np.array([[1, 2], [3, 4]])
# Creating a 3D array
arr_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
```

20.2 Difference between NumPy arrays and lists

NumPy arrays offer better performance, more functionality, and memory efficiency compared to Python lists. Unlike lists, NumPy arrays are homogeneous, allowing for optimized operations.

20.3 Higher dimension arrays

Higher-dimensional arrays are useful for applications like image processing or deep learning.

```
# Creating a three-dimensional array
a5 = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4,5,6]]])
print(a5)
# Creating a higher-dimensional array (4D)
high_dim_array = np.ones((2, 3, 4, 5))
```

20.4 Array Properties/Attributes

NumPy arrays have several attributes that provide information about their structure and data.

```
arr = np.array([1, 2, 3, 4])
print(arr.ndim) # Number of dimensions
print(arr.shape) # Shape of the array (dimensions per axis)
print(arr.size) # Total number of elements
print(arr.dtype) # Data type of elements
print(arr.itemsize) # Size of each element in bytes
```

20.5 Accessing Array Elements

Accessing elements in a NumPy array uses zero-based indexing, similar to Python lists.

```
# Accessing elements in a 1D array
print(arr[0])
# Accessing elements in a 2D array
print(arr_2d[0, 1])
# Accessing elements in a 3D array
print(arr_3d[0, 1, 1])
```

20.6 Conditional Indexing

Conditional indexing allows for filtering elements based on a condition.

```
arr = np.array([10, 20, 30, 40, 50])
print(arr[arr > 30]) # Retrieves elements greater than 30
arr[arr > 30] = -1 # Updates elements greater than 30 to -1
print(arr)
```

20.7 Mutability

NumPy arrays are mutable, meaning you can modify their contents.

```
arr[0] = 10 # Modifying an element
print(arr)
```

20.8 Referencing

Assigning an array to another variable creates a reference, not a copy. Changes in the new variable affect the original.

```
ref = arr
ref[1] = 20
print(arr) # Original array is also modified
```

20.9 Reshaping

You can reshape arrays to change their structure without altering the data.

```
arr = np.ones((2, 3, 4), dtype=np.int16)
print("Before reshape:")
```

```
print(arr)
arr = arr.reshape(6, 4)
print("After reshape:")
print(arr)
```

20.10 Data Types

You can explicitly change the data type of array elements using the `astype` method.

```
arr_float = arr.astype(float)
print(arr_float.dtype)
print(arr_float)
```

Initialization

20.11 Zeros and Ones

You can initialize arrays filled with zeros or ones, useful for default values.

```
zeros = np.zeros((2, 3))
ones = np.ones((2, 3))
arr_eye = np.eye(4) # Generates a square identity matrix
```

20.12 Range

NumPy offers functions to create arrays with numerical ranges.

```
# Creating a range of numbers
arr_range = np.arange(0, 10, 2)
# Creating equally spaced numbers
linspace = np.linspace(0, 1, 5) # Five numbers between 0 and 1
```

20.13 Operations

Arithmetic

Arithmetic operations in NumPy are performed elementwise.

```
arr1 = np.array([10, 11, 12, 13, 14, 15])
arr2 = np.array([20, 21, 22, 23, 24, 25])
print(arr1 + arr2) # Addition
print(arr1 - arr2) # Subtraction
print(arr1 * arr2) # Multiplication
print(arr1 / arr2) # Division
```

20.14 Matrix

Matrix Multiplication

```
m1 = np.array([[1, 2], [3, 4]])
m2 = np.array([[5, 6], [7, 8]])
# Element-wise multiplication
m_mul = np.multiply(m1, m2)
# Matrix multiplication
m_dot = np.dot(m1, m2) # Alternatively, use m1 @ m2
```

20.15 Transpose

```
matrix = np.array([[1, 2, 3], [4, 5, 6]])
m_transpose = np.transpose(matrix)
print(m_transpose)
```

20.16 Determinant

The determinant of a square matrix is calculated using `numpy.linalg.det`

```
from numpy.linalg import det
matrix_sq = np.array([[1, 2], [3, 4]])
print(det(matrix_sq))
```

20.17 Universal Functions

Universal functions (ufuncs) are vectorized operations applied element-wise.

```
arr = np.array([1, 4, 9, 16])
sqrt_result = np.sqrt(arr) # Square root of each element
sin_result = np.sin(arr) # Sine of each element
```

20.18 Understanding the `axis` Parameter

In many NumPy operations (like `np.sum()`, `np.mean()`), the `axis` parameter specifies the dimension to perform the operation along. For a 2D array:

- `axis=0` performs the operation **column-wise** (collapsing the rows).
- `axis=1` performs the operation **row-wise** (collapsing the columns).

---

21. Random Module

NumPy's random module provides functions to generate random numbers.

```
# Generating random numbers between 0 and 1
random_array = np.random.rand(2, 3)

# Generating random integers within a range
random_integers = np.random.randint(0, 10, size=(3, 3))
```

21.1 Distribution

`numpy.random.rand(d0, d1, ..., dn)`

```
# Uniform Distribution: Generates random numbers to form an array
arr = np.random.rand(10000) # This generates 10,000 numbers between 0 and 1
```

`numpy.random.randn(d0, d1, ..., dn)`

```
# Generates numbers from a standard normal distribution
# (mean = 0, std dev = 1)
arr = np.random.randn(10000) # This generates 10,000 numbers from the normal distribution
```

`numpy.random.exponential(scale, size=None)`

```
# Here scale is the inverse of the rate parameter
arr = np.random.exponential(1, 10000)
```

21.2 Seed

Setting a seed ensures that the random numbers generated are reproducible.

```
numpy.random.seed(seed) # seed is an integer value
np.random.seed(182)
print(np.random.rand(10))
```

---

22. Statistical Analysis

22.1 Mean

The mean (average) is calculated using `numpy.mean`

```
arr = np.array([1, 2, 3, 4, 5])
mean_val = np.mean(arr)
```

22.2 Median and Mode

The median is the middle value, while the mode is the most frequent value.

```
arr = np.array([1, 2, 3, 4, 5])
median_val = np.median(arr)
from scipy.stats import mode
mode_val = mode(arr)
```

22.3 Standard deviation and Variance

Standard deviation measures spread, while variance measures the squared spread.

```
arr = np.array([1, 2, 3, 4, 5])
std_val = np.std(arr)
var_val = np.var(arr)
```

22.4 Statistical Range

The range is the simplest measure of statistical dispersion. It is the difference between the maximum and minimum values in a dataset.

- **Formula:** Range = Maximum Value - Minimum Value

22.5 Visualization with Matplotlib & Seaborn

NumPy arrays can represent image data or be used in plots with libraries like Matplotlib and Seaborn.

- **Histograms (plt.hist):** Excellent for visualizing the **distribution** of a single numerical variable.
- **Bar Charts (plt.bar):** Used for **comparing** quantities across different categories.
- **Scatter Plots (plt.scatter):** Used to observe the **relationship** between two numerical variables.

Original Example:

```
import matplotlib.pyplot as plt
# Generate random image data
image = np.random.random((256, 256, 3))
```

```
plt.imshow(image)
plt.show()
```

22.6 Introduction to Seaborn :

Seaborn is a higher-level library based on Matplotlib that simplifies creating complex statistical plots.

- **Example (seaborn.scatterplot):** This function creates a scatter plot and allows for adding more dimensions. The hue parameter is commonly used to assign colors to points based on a third categorical variable.

```
import seaborn as sns
# Assumes 'df' is a Pandas DataFrame
# with columns 'x_data', 'y_data', and 'category'
sns.scatterplot(data=df, x='x_data', y='y_data', hue='category')
```

22.7 Visualizing Algorithm Efficiency (Required Addition)

Data visualization can be used to understand algorithm efficiency. By plotting the input size (n) on the x-axis and the computation time (or number of operations) on the y-axis, one can visually confirm an algorithm's complexity class (e.g., a steep curve for $O\left(n^2\right)$ vs. a much flatter curve for $O(nlogn)$).

22.8 Basic Probability Concepts (Required Addition)

Probability is the measure of the likelihood that an event will occur.

- **Basic Formula:**

$$P(Event) = \frac{Number\ of\ Favorable\ Outcomes}{Total\ Number\ of\ Possible\ Outcomes}$$

- **Conditional Probability:** The probability of event A occurring *given that* event B has already occurred.

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

22.9 Hypothesis Testing

Hypothesis testing is a formal statistical method used to make decisions based on experimental data. It allows us to determine if there is enough evidence to support a specific claim about a population.

- **Null Hypothesis (H_0):** This is the default assumption or the statement of "no effect." It represents the status quo that we are trying to find evidence against (e.g., "The average student GPA is 3.0").
- **Alternative Hypothesis** $\left(H_a \text{ or } H_1\right)$ **:** This is the claim we are testing for. It's what we will conclude if we find enough evidence to reject the null hypothesis (e.g., "The average student GPA is not 3.0").
- **One-Tailed vs. Two-Tailed Tests:**
  - **One-tailed:** The alternative hypothesis has a specific direction (e.g., the average GPA is *greater than* 3.0, or the average GPA is *less than* 3.0). You are only testing for a change in one direction.
  - **Two-tailed:** The alternative hypothesis has no direction; it simply states that there is a difference (e.g., the average GPA is *not equal to* 3.0). You are testing for a change in both directions (greater than or less than).
- **Significance Level (α):** This is a pre-determined threshold for how much evidence we need to reject the null hypothesis. It represents the maximum risk we are willing to take of making a "false positive" error (rejecting H0 when it's actually true). **The most common value for α is 0.05 (or 5%).**
- **p-value:** This is the probability of observing our data (or data more extreme) if the null hypothesis were true. It's the key result of a hypothesis test.
  - **Decision Rule:** If the **p-value is less than the significance level (α)**, we **reject the null hypothesis**. This means our result is statistically significant.
  - If $p - value \geq \alpha$, we **fail to reject the null hypothesis**.

Common Formulas

- **One-Sample t-test:** Used to test if a sample mean $(\bar{x})$significantly different from a known or hypothesized population mean (μ).

$$t = \frac{\bar{x} - \mu}{s/\sqrt{n}}$$

Where $\bar{x}$ *is the sample mean, μ is the population mean, s is the sample standard deviation, and n is the sample size.*

- **Chi-Square (χ^2) Test:** Used to test if there is a significant association between two categorical variables by comparing observed frequencies (O) to expected frequencies (E).

$$\chi^2 = \sum \frac{(O-E)^2}{E}$$

Where O is the observed frequency and E is the expected frequency for each category.

---

23. Pandas DataFrame

23.1 Accessing Multiple Columns Concept

Pandas allows users to access multiple columns by passing a list of column names to the DataFrame. This technique is particularly useful for isolating specific subsets of data for further analysis or visualization.

23.2 Introduction to Pandas DataFrames

- Pandas provides a powerful data structure called DataFrame, similar to a table or spreadsheet.
- It allows for easy manipulation and analysis of structured data.

23.3 Creating a DataFrame:

```
import pandas as pd
import numpy as np
student_data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': np.random.randint(18, 22, size=5),
    'Major': ['Computer Science', 'Physics', 'Mathematics', 'Engineering', 'Biology'],
    'GPA': np.random.uniform(2.5, 4.0, size=5).round(2)
}
df = pd.DataFrame(student_data)
print(type(df)) # Output: <class 'pandas.core.frame. DataFrame'>
print(df)
```

23.4 Basic Information About the DataFrame:

```
df.info()
```

- **Displays:**
  - Number of rows and columns.
  - Data types of each column.
  - Memory usage.

23.5 Accessing Data

Accessing Columns:

```
print(df['Name']) # Access using column name as key
print(df.Age) # Access using dot notation
```

Accessing Rows:

```
print(df.iloc[0]) # Access the first row by index
```

- **iloc:** Access rows by their numerical index.
- **loc:** Access rows by labels (if available).

Summary Statistics:

- Pandas provides functions to compute basic statistics for numerical columns:

```
print(df.describe()) # Summary statistics for numerical columns
```

Additional Pandas Topics: Adding New Columns:

```
df['Graduation Year'] = 2025
print(df)
```

23.6 Filtering Data:

```
print(df[df['GPA'] > 3.5]) # Students with GPA greater than 3.5
```

Handling Missing Data:

- Fill missing values:

```
df['GPA'].fillna(df['GPA'].mean(), inplace=True)
```

- Drop rows with missing values:

```
df.dropna(inplace=True)
```

23.7 Slicing Rows

Concept

Row slicing allows you to extract specific rows from a DataFrame based on their index positions.

Syntax

```
df.iloc[start:stop]
```

- start: The starting index (inclusive).
- stop: The ending index (exclusive).

Example

```
print(df.iloc[1:4]) # Retrieves rows with index 1, 2, and 3.
```

23.8 Accessing First Rows

Accessing First Rows

Concept

The head() function retrieves the first few rows of the DataFrame.

Syntax

df.head(n)

- n: The number of rows to retrieve (default is 5).

Example

```
print(df.head(2)) # Retrieves the first 2 rows.
```

23.9 Accessing Last Rows

Concept

The tail() function retrieves the last few rows of the DataFrame.

Syntax df.tail(n)

- n: The number of rows to retrieve (default is 5).

Example

```
print(df.tail(2)) # Retrieves the last 2 rows.
```

23.10 Statistical Analysis on Data

Concept

Pandas offers built-in methods for performing statistical computations and summarizing data, which are crucial for understanding data distributions and trends.

Common Functions

1. **describe()**: Provides a statistical summary of numerical columns, including count, mean, standard deviation, minimum, and quartiles.
2. **mean()**: Calculates the average value of a column.

3. `min()`: Identifies the smallest value in a column.
4. `max()`: Identifies the largest value in a column.

Querying (Concept)

Querying enables you to filter rows in a DataFrame based on specific conditions. This technique is pivotal for isolating relevant data.

Syntax

```
df[condition]
```

- **condition**: A logical condition (e.g., `df['Age'] > 20`) used to filter rows.

Examples

1. Student with Highest GPA:

```
# Retrieves the row where the "GPA" column has the maximum value.
df[df['GPA'] == df['GPA'].max()]
```

2. Details of a Specific Student (e.g., Bob):

```
# Filters rows where the "Name" column matches "Bob".
df[df['Name'] == 'Bob']
```

3. Specific Columns for a Condition:

```
# Retrieves the "Major" and "GPA" columns for the student named "Bob".
df[df['Name'] == 'Bob'][['Major', 'GPA']]
```

4. Students with a Specific Age:

```
# Filters rows where the "Age" column equals 18.
df[df['Age'] == 18]
```

23.12 Reading Data from CSV

Concept

The `read_csv()` function reads data from a CSV file and loads it into a DataFrame.

Syntax

`pd.read_csv(filepath)`

23.13 Getting DataFrame Information

Concept

- **`info()`**: Displays metadata (column names, data types, non-null values). `df.info()`
- **`describe()`**: Provides a statistical summary of numerical columns. `df.describe()`

23.14 Finding Specific Values in DataFrame

Concept

Conditional indexing enables you to locate rows that meet specific criteria by setting a logical condition inside the selection brackets.

Example

1. Day with Maximum Temperature:

```
# This code finds the row(s) where the 'Temperature' column equals its own maximum value.
df[df['Temperature'] == df['Temperature'].max()]
```

- **Explanation:** This retrieves the entire row from the DataFrame where the "Temperature" column has the maximum value.

23.14 Data Aggregation with .groupby()

The `.groupby()` method allows you to split a DataFrame into groups based on some criteria, apply a function (like `.sum()` or `.mean()`) to each group, and combine the results. **Example:**

```
# Given the student DataFrame 'df' from section 23.3
# Find the average GPA for each 'Major'
major_gpa = df.groupby('Major')['GPA'].mean()
print(major_gpa)
```

---

24. Object-Oriented Programming (OOP)

24.1 Classes and Objects

- **Description:**
  - **Class:** A blueprint that defines attributes (data) and methods (functions) to represent real-world entities.

- **Object:** A concrete instance of a class with actual values.
- Think of a class as a template and an object as a specific example built from that template.

- **Example:**

```
# Define a class named 'Car'
class Car:
    # The __init__ method initializes new objects with attributes.
    def __init__(self, brand, model):
        self.brand = brand # Attribute to store the brand
        self.model = model # Attribute to store the model

    # A method that returns a string with car information.
    def get_info(self):
        return f"{self.brand} {self.model}"

# Create objects (instances) of the Car class
car1 = Car("Toyota", "Corolla")
car2 = Car("Honda", "Civic")

print(car1.get_info()) # Output: Toyota Corolla
print(car2.get_info()) # Output: Honda Civic
```

- **Explanation:**

- The `Car` class serves as a blueprint for creating car objects.
- The `__init__` method is called automatically when a new object is created and sets up its `brand` and `model` attributes.
- `car1` and `car2` are objects created from the `Car` class.
- The `get_info()` method returns the car's details in a formatted string.

24.2 Encapsulation

- **Description:** Encapsulation is the process of wrapping data (attributes) and methods (functions) into a single unit (a class) and controlling access to that data. This helps protect the data from being modified directly from outside the class.

- **Example:**

```
class BankAccount:
    def __init__(self, account_number, balance):
        self.account_number = account_number
        # A double underscore denotes a private attribute
        self.__balance = balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            return self.__balance
```

```
    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            return self.__balance
        else:
            return "Insufficient funds or invalid amount"

account = BankAccount("12345", 1000)
print(account.deposit(500))  # Output: 1500
print(account.withdraw(300)) # Output: 1200
# print(account.__balance) # This would cause an AttributeError
```

- **Explanation:**
  - The `BankAccount` class encapsulates the account data and operations.
  - The attribute `__balance` is marked as private. It should not be directly accessed from outside the class.
  - Methods `deposit()` and `withdraw()` provide controlled ways to modify the balance, ensuring invalid operations are handled safely.

24.3 Inheritance

- **Description:** Inheritance allows a new class (the **child class**) to acquire the properties and behaviors (attributes and methods) of an existing class (the **parent class**). This promotes code reusability and a hierarchical organization.

- **Example:**

```
# Parent class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return "Animal makes a sound"

# Child class that inherits from Animal
class Dog(Animal):
    # Override the speak method to provide specific behavior
    def speak(self):
        return f"{self.name} barks"

dog = Dog("Buddy")
print(dog.speak()) # Output: Buddy barks
```

- **Explanation:**
  - The `Animal` class defines a basic template.

- The `Dog` class inherits from `Animal`, automatically getting the `name` attribute and its methods.
- The `Dog` class then **overrides** the `speak()` method to provide behavior specific to dogs.

24.4 Polymorphism

- **Description:** Polymorphism means "many shapes" and allows methods to have different behaviors in different classes. A method name can work differently depending on the object that calls it.

- **Example:**

```
class Bird:
    def move(self):
        return "Flies in the sky"

class Penguin(Bird):
    # Overriding the move method for a different behavior
    def move(self):
        return "Swims in the water"

def show_movement(animal):
    print(animal.move())

sparrow = Bird()
pingu = Penguin()

show_movement(sparrow) # Output: Flies in the sky
show_movement(pingu)   # Output: Swims in the water
```

- **Explanation:**

  - Both `Bird` and `Penguin` have a `move()` method, but their implementations differ.
  - The `show_movement()` function demonstrates polymorphism by calling the same method (`.move()`) on different objects and getting different results.

24.5 Abstraction

- **Description:** Abstraction is about hiding complex implementation details and showing only the necessary features of an object. In Python, this is typically achieved through Abstract Base Classes (ABC).

- **Example:**

```
from abc import ABC, abstractmethod
```

```python
# Abstract class that cannot be instantiated directly
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass # Must be implemented by subclasses

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    # Concrete implementation of the abstract method
    def area(self):
        return self.width * self.height

rect = Rectangle(5, 4)
print(rect.area()) # Output: 20
```

- **Explanation:**
  - The `Shape` class is an abstract class with an abstract method `area()`.
  - Abstract methods must be defined in any subclass that inherits from `Shape`.
  - The `Rectangle` class provides a specific implementation for `area()`, fulfilling the contract of the abstract class.

24.6 Method Overloading vs. Overriding

- **Description:**
  - **Method Overloading:** Involves having multiple methods with the same name but different parameters. Python simulates this with default parameters or variable-length arguments.
  - **Method Overriding:** Occurs when a child class provides a new implementation for a method already defined in its parent class.
- **Example:**

```python
# Method Overloading (using default arguments)
class MathOperations:
    def add(self, a, b, c=0): # c is optional
        return a + b + c

math_obj = MathOperations()
print(math_obj.add(5, 10))      # Output: 15 (c defaults to 0)
print(math_obj.add(5, 10, 20))  # Output: 35
```

```
# Method Overriding
class Parent:
    def show(self):
        return "Parent class method"

class Child(Parent):
    # Overriding the show method of the Parent class
    def show(self):
        return "Child class method"

child_obj = Child()
print(child_obj.show()) # Output: Child class method
```

- **Explanation:**
  - In overloading, the add() method handles both two and three arguments.
  - In overriding, the Child class redefines the show() method from Parent, so it produces a different output.

24.7 Access Modifiers

- **Description:** Access modifiers in Python are conventions that control the visibility of class attributes and methods.
  - **Public:** No special symbol; accessible from anywhere.
  - **Protected:** Prefixed with a single underscore (_). A convention suggesting it should not be accessed directly outside the class or its subclasses.
  - **Private:** Prefixed with two underscores (__). The attribute name is "mangled," making it not easily accessible from outside the class.
- **Example:**

```
class Example:
    def __init__(self):
        self.public = "Public Attribute"
        self._protected = "Protected Attribute"
        self.__private = "Private Attribute"

obj = Example()
print(obj.public)       # Output: Public Attribute
print(obj._protected)   # Output: Protected Attribute (access is possible but discouraged)
# print(obj.__private)    # This would raise an AttributeError
```

- **Explanation:**
  - Public attributes are part of the class's public API.
  - Protected attributes are a hint to programmers that they are for internal use.
  - Private attributes are hidden to protect sensitive data and enforce encapsulation.

24.8 Summary Table:

Concept	Description	Key Example
Class & Object	Blueprint for objects, an instance of the class	`Car("Toyota", "Corolla")`
Encapsulation	Bundling data and methods, restricting direct access	`BankAccount` with `__balance`
Inheritance	Child class reusing and extending features of a parent	`Dog` inheriting from `Animal`
Polymorphism	Same method name acting differently	`move()` in `Bird` vs. `Penguin`
Abstraction	Hiding complex details, exposing necessary parts	Abstract `Shape` class
Overloading & Overriding	Overloading: Same name, diff. params; Overriding: Redefining method	`add()` method & `Child.show()`
Access Modifiers	Defines visibility: public, protected, private	`public`, `_protected`, `__private`

Module B

1. Regression: Linear and Polynomial

1.1 Linear Regression

- **Equation:**

$$y = mx + c$$

- **Properties:**
  - Assumes a linear relationship between x and y.
  - Limited for complex patterns.

1.2 Polynomial Regression

- **Equation:**

$$y = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

- **Properties:**
  - Fits non-linear data by adding polynomial terms.
  - Degree n determines model complexity.
  - More flexible but prone to overfitting.

1.3 Overfitting vs. Underfitting

- **1.3.1 Overfitting**
  - **Definition:** Model captures noise and patterns specific to training data, resulting in poor generalization.
  - **Causes:** High model complexity (e.g., large degree in polynomial regression).
  - **Fixes:** Reduce model complexity, use regularization, increase training data.
- **1.3.2 Underfitting**
  - **Definition:** Model is too simple to capture underlying patterns in the data.
  - **Causes:** Low model complexity (e.g., linear regression for non-linear data).
  - **Fixes:** Increase model complexity, use higher-degree polynomial terms, improve feature engineering.

1.4 Logistic Regression

Logistic Regression is a fundamental algorithm used for **binary classification** (e.g., Spam/Not Spam, 0/1).

- **Concept:** It models the probability that an input belongs to a specific class.
- **Sigmoid Function:** It uses the sigmoid (or logistic) function to map any real-valued number into a probability between 0 and 1.

- **Formula:**

$$f(x) = \frac{1}{1+e^{-x}}$$

- **Interpreting Output:**
  - The model outputs a probability. Methods like `.predict_proba()` in scikit-learn return this value (e.g., `[0.1, 0.9]`, meaning 90% chance of class 1).
  - A threshold (typically 0.5) is used to convert this probability into a hard class label (e.g., since 0.90.5, predict class 1).

1.5 L1 & L2 Regularization

Regularization is a technique to prevent overfitting by adding a penalty to the loss function based on the size of the model's coefficients (w_i).

- **L2 Regularization (Ridge):**
  - **Formula:**

$$Loss + \lambda \sum w_i^2$$

  - **Effect:** Adds a "squared magnitude" penalty. It shrinks large coefficients, but it **does not make them exactly zero**.
- **L1 Regularization (Lasso):**
  - **Formula:**

$$Loss + \lambda \sum |w_i|$$

  - **Effect:** Adds an "absolute value" penalty. It can shrink some coefficients to **exactly zero**, which makes it useful for **automatic feature selection** by effectively removing unimportant features from the model.

---

2. Principal Component Analysis (PCA) & t-SNE

The Big Idea: Finding the Best Shadows

Imagine you have a complex 3D object, but you can only draw 2D pictures (shadows) of it. Some shadows will be very informative, showing the object's main shape, while others will be unhelpful squiggles.

> PCA is a mathematical technique for finding the "best shadows" for your data. It reduces a complex, high-dimensional dataset into a simpler, low-dimensional one while keeping the most important information.

It solves the "curse of dimensionality" by transforming the original features into a new, smaller set of features called **Principal Components**. These new features are ordered by how much information (or **variance**) from the original data they capture.

2.1 How PCA Works: A Step-by-Step Guide

The process involves finding new axes for your data that align with the directions of maximum variance.

2.1.1 Step 1: Standardize the Data

Before starting, all features must be on the same scale. If one feature is in kilograms and another is in millimeters, the variance of the millimeter feature will dominate. Standardization converts all features to have a **mean of 0** and a **standard deviation of 1**.

2.1.2 Step 2: Compute the Covariance Matrix

This is the core of PCA. The covariance matrix is a square matrix that summarizes the relationships between all pairs of features in your data.

- The elements on the **main diagonal** represent the **variance** of each feature (how spread out it is).
- The **off-diagonal** elements represent the **covariance** between pairs of features (how they move together).

The formula for the sample covariance matrix is:

$$C = \frac{1}{n-1} X^T X$$

where: C = Covariance matrix, X = Standardized data matrix, n = Number of samples

2.1.3 Step 3: Find Eigenvectors and Eigenvalues

Once we have the covariance matrix, we calculate its eigenvectors and eigenvalues.

- **Eigenvectors:** These are the directions of the new axes where the data has the most variance. Each eigenvector represents a **Principal Component**. The first principal component (PC1) points in the direction of the highest variance.
- **Eigenvalues:** These are numbers that tell you the *amount* of variance captured by each eigenvector.

The eigenvector with the highest eigenvalue is your most important Principal Component. We sort the eigenvectors by their corresponding eigenvalues, from highest to lowest.

2.1.4 Step 4: Select Principal Components

We decide how many of the new axes (principal components) to keep, often by looking at the **explained variance**. For example, if PC1 explains 70% of the variance and PC2 explains 25%, we might keep just those two components to capture 95% of the total information.

2.1.5 Step 5: Transform the Data

The final step is to project the original standardized data onto the new axes (the principal components we decided to keep).

$$Z = X \cdot W$$

where:
Z = The new, transformed data ($n \times k$ matrix)
X = The original standardized data ($n \times d$ matrix)
W = The matrix of k chosen eigenvectors ($d \times k$ matrix)

2.1.6 Summary & Key Properties

- PCA reduces dimensionality while retaining the maximum amount of variance.
- It works by finding a new set of orthogonal (uncorrelated) axes called principal components.
- It is highly sensitive to the scale of the data, which is why **standardization is a required first step**.
- It is a powerful tool for noise reduction, data visualization, and preparing data for other machine learning algorithms.

2.2 t-SNE (t-Distributed Stochastic Neighbor Embedding)

t-SNE is a powerful **visualization technique** for high-dimensional data. It is a non-linear dimensionality reduction tool, not a clustering algorithm.

- **Local Neighborhoods:** Its primary goal is to preserve the **local structure** of the data. This means points that are close to each other in high-dimensional space remain close in the final 2D or 3D plot.
- **Workflow:** t-SNE is computationally expensive. A common workflow is to first reduce dimensions with **PCA** (e.g., to 30-50 dimensions) and then use **t-SNE** to reduce it further to 2 or 3 dimensions for plotting.

---

3. PageRank

3.1 Graph Theory Basics (for PageRank)

To understand PageRank, some basic graph terms are necessary.

- **Directed Graph:** A graph where edges have a direction, representing one-way links (e.g., Page A links to Page B, but not necessarily vice-versa).
- **Node (or Vertex):** Represents an entity (e.g., a web page).
- **Edge:** Represents a connection or link between two nodes.
- **Indegree:** The number of **incoming** edges a node has. In PageRank, a high indegree (many pages linking *to* a page) is a primary indicator of importance.

3.2 Process

PageRank is an algorithm that assigns a numerical "rank" to each node in a directed graph, estimating its importance.

1. **Graph Representation:** Represent the web as a directed graph.
2. **Initial Rank Assignment:** Assign an equal rank to all pages (e.g., $1/N$).
3. **Rank Update:** Iteratively update the rank of each page based on the ranks of the pages that link to it. A page's rank is "shared" equally among all its outgoing links.
4. **Convergence:** Continue until the rank values stabilize.

3.3 Components and Their Effect

- **Nodes (Web Pages):** Pages to rank.
- **Edges (Hyperlinks):** Connections between pages.
- **Outlinks (Outgoing Links):** Pages share their rank among all pages they link to.
- **Damping Factor (d)**: A factor (usually 0.85) is used to represent a "**random surfer**" who stops clicking links and jumps to a random page. This solves problems with pages that have no outgoing links (dead ends) or groups of pages that only link to each other (rank sinks).
- **Iterations:** Repeated calculations to redistribute ranks until they converge.

3.3.1 The PageRank Formula

The rank of a given page **A** is calculated using the ranks of all the pages that link to it. The formula incorporates both the probability of arriving at the page by following links and by randomly jumping to it.

The complete PageRank formula is:

$$PR(A) = \frac{1-d}{N} + d\sum_{i=1}^{n} \frac{PR(T_i)}{L(T_i)}$$

Where:

- $PR(A)$ is the PageRank of the page **A** we are calculating.
- d is the **damping factor**, typically set to **0.85**.
- N is the total number of pages in the collection.

- T_i are the pages that link to page **A**.
- $PR(T_i)$ is the PageRank of a linking page T_i.
- $L(T_i)$ is the number of outgoing links on page T_i.

3.4 Worked Numerical Example (with Damping Factor)

Problem: 4 pages (A, B, C, D) with the following links:

- $A \rightarrow B$
- $B \rightarrow C$
- $C \rightarrow A, D$
- $D \rightarrow C$

Parameters:

- Total pages $N = 4$.
- Damping factor $d = 0.85$.
- The number of outgoing links for each page is: $L(A) = 1$, $L(B) = 1$, $L(C) = 2$, $L(D) = 1$.
- **Initial Rank**: Each page is assigned an equal rank of $1/N = 1/4 = 0.25$.

Iteration 1:

We apply the full formula to update the rank of each page. The "random jump" probability for each page is

$$\frac{1-d}{N} = \frac{1-0.85}{4} = \frac{0.15}{4} = 0.0375$$

- **Calculate PR(A):** Page A is linked by Page C.

$$PR(A) = 0.0375 + 0.85 \times \left(\frac{PR(C)}{L(C)}\right) = 0.0375 + 0.85 \times \left(\frac{0.25}{2}\right) = 0.0375 + 0.10625 = 0.14375$$

- **Calculate PR(B):** Page B is linked by Page A.

$$PR(B) = 0.0375 + 0.85 \times \left(\frac{PR(A)}{L(A)}\right) = 0.0375 + 0.85 \times \left(\frac{0.25}{1}\right) = 0.0375 + 0.2125 = 0.25$$

- **Calculate PR(C):** Page C is linked by Page B and Page D.

$$PR(C) = 0.0375 + 0.85 \times \left(\frac{PR(B)}{L(B)} + \frac{PR(D)}{L(D)}\right) = 0.0375 + 0.85 \times \left(\frac{0.25}{1} + \frac{0.25}{1}\right) = 0.0375 + 0.425 = \mathbf{0.4625}$$

- **Calculate PR(D):** Page D is linked by Page C.

$$PR(D) = 0.0375 + 0.85 \times \left(\frac{PR(C)}{L(C)}\right) = 0.0375 + 0.85 \times \left(\frac{0.25}{2}\right) = 0.0375 + 0.10625 = 0.14375$$

After the first iteration, the new ranks are A=0.144, B=0.25, C=0.463, and D=0.144. This iterative process would continue with these new values until the ranks converge and no longer change significantly.

---

4. Neural Networks

4.1 Foundations of the Perceptron

- **Prediction:**

$$y = sign(w \cdot x + b)$$

- **y:** Output label (+1 or -1)
- **w:** Weight vector
- **x:** Input feature vector
- **b:** Bias
- **Limitation (Required Addition):** A Perceptron is a single-layer neural network. Its primary limitation is that it can only learn and classify **linearly separable data**. It cannot solve problems where the classes are not separable by a straight line (e.g., the XOR problem).

4.2 Enhancing the Perceptron with Learning Algorithms

- **Weight Update Rule:**

$$w_{new} = w_{old} + \Delta w \quad where \quad \Delta w = \eta\left(y_{true} - y_{pred}\right)x$$

- η: Learning rate
- y_{true}: True label
- y_{pred}: Predicted label

4.3 Neuron: Basic Building Block

A neuron performs two main operations:

1. **Net Operation:** Computes the weighted sum of inputs.
2. **Out Operation:** Applies an activation function to produce the output.

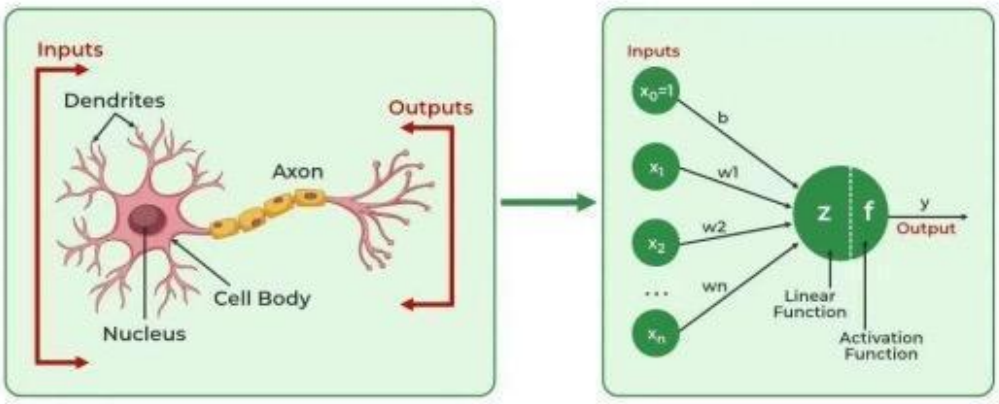


Figure 1: Neuron

4.4 Components of a Neuron

The core elements of a neuron are:

- **Inputs ($x_1, x_2, ..., x_n$):** Values that represent features or outputs from the previous layer.
- **Weights ($w_1, w_2, ..., w_n$):** Coefficients that signify the importance of each input.
- **Bias (b):** An additional value that shifts the weighted sum to enhance flexibility.
- **Activation Function (f):** A function that introduces non-linearity to the model, enabling it to learn complex patterns.
- **Output (y):** The final result computed by applying the activation function to the weighted sum of inputs and bias.

4.5 Mathematical Representation

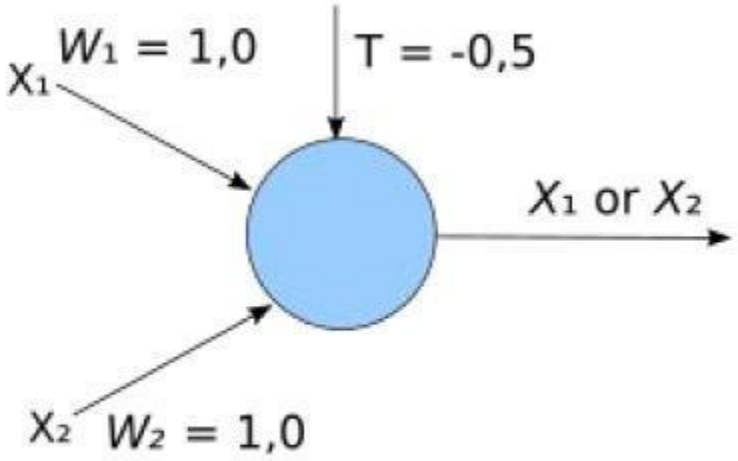


Figure 2: Components of a Neuron

To compute the output of a neuron, first calculate the weighted sum of inputs and bias:

$$Net\ Input\ (z) = \left(\sum_{i=1}^{n} w_i x_i\right) + b = w \cdot x + b$$

This value is then passed through the activation function:

$$Output\ (y) = f(z)$$

In words: The neuron's output is the result of applying the activation function f to the sum of the weighted inputs ($w \cdot x$) and the bias (b).

4.6 Structure of a Neural Network

A neural network consists of three main layers:

- **Input Layer:** Accepts raw data and passes it to the next layer.
- **Hidden Layer(s):** Processes inputs using weights and biases, and learns patterns.
- **Output Layer:** Produces the final predictions.

Each neuron performs two steps:

1. Compute the weighted sum of inputs (net operation).
2. Apply the activation function to get the output (out operation).

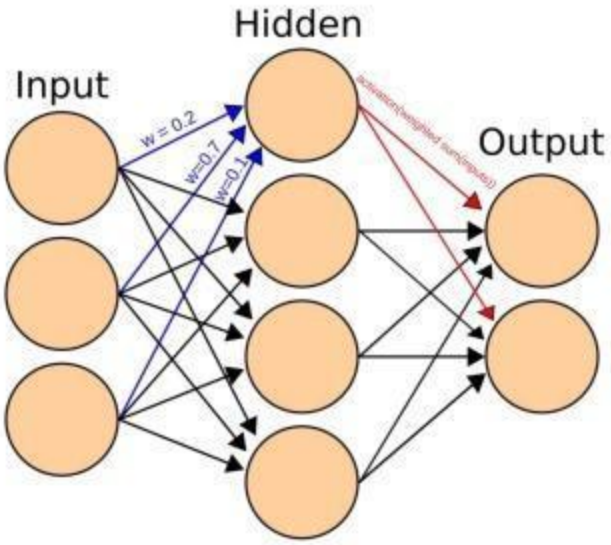


Figure 3: Structure of an Artificial Neural Network

---

5. Forward and Backward Propagation

5.1 Overview

1. **Forward Pass:** Compute the outputs based on the current weights.
2. **Backward Pass:** Calculate gradients (the rate of change of the error) and propagate errors backward using the chain rule of differentiation.

5.1.1 Gradient Descent Variants

Gradient Descent is the optimization algorithm used to update weights (w). The standard update rule is: $w_{new} = w_{old} - \eta \nabla L$, where η is the learning rate and ∇L is the gradient of the loss.

- **Batch Gradient Descent:** Calculates the gradient using the **entire** training dataset. Very accurate but computationally slow.
- **Stochastic Gradient Descent (SGD):** Calculates the gradient using **one** training sample at a time. Fast and can escape local minima, but the updates are noisy.
- **Mini-Batch Gradient Descent:** A compromise. Calculates the gradient using a **small batch** (e.g., 32 or 64 samples) of data. This is the most common variant, balancing speed and stability.

5.2 Forward Pass

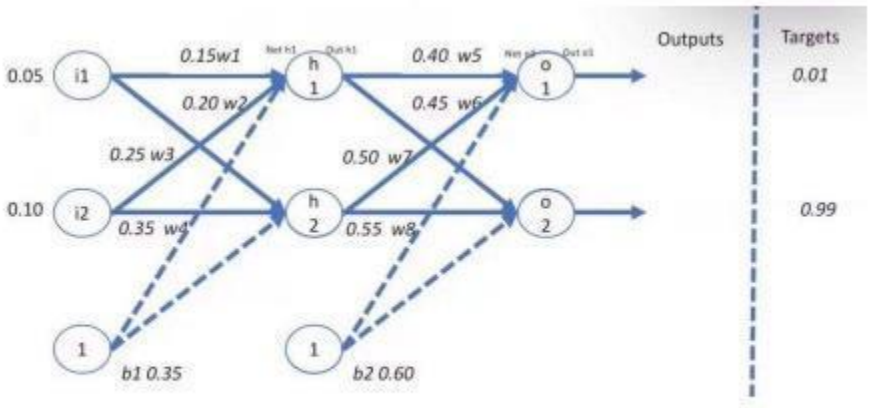


Figure 4: Example of Forward Propagation in a Neural Network

1. **Weighted Sum of Inputs:** For a neuron, the net input is calculated as the weighted sum of the inputs to the neuron, plus a bias term:

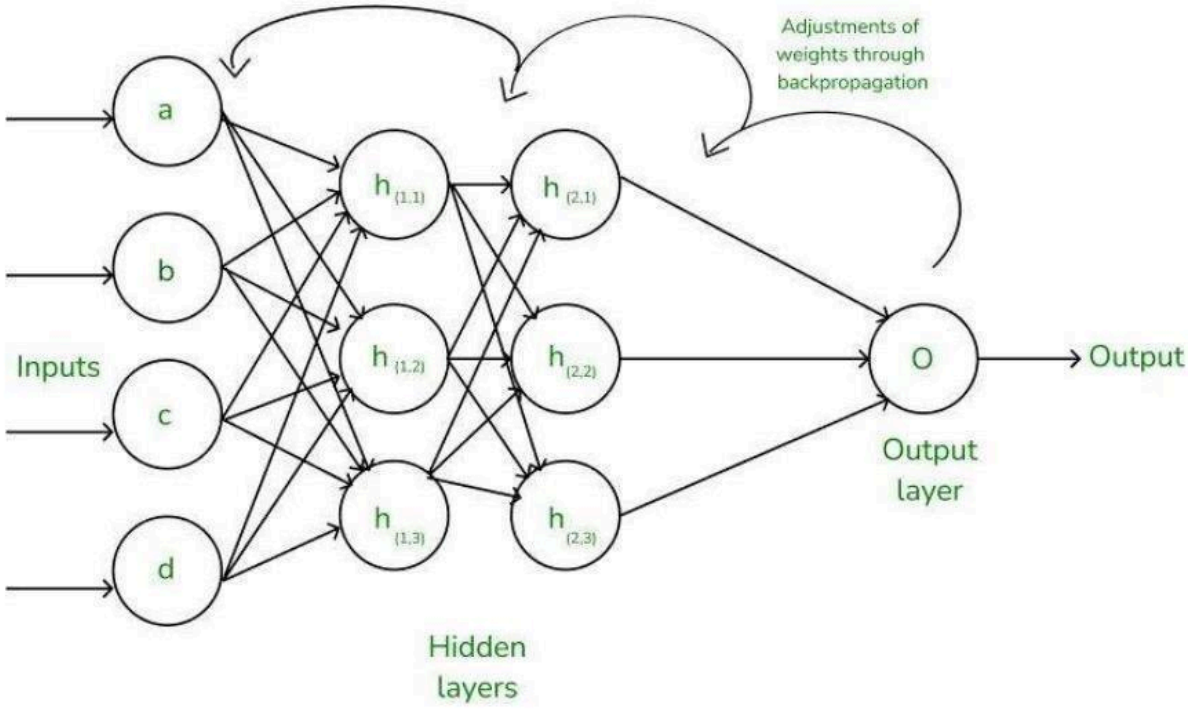
$$net_j = \sum_i w_{ij} x_i + b_j$$

where w_{ij} are the weights, x_i are the inputs, and b_j is the bias.

2. **Activation Function:** The output of the neuron is then obtained by applying an activation function to the net input.

5.3 Backward Pass

Figure 5: Backward Propagation in an Artificial Neural Network



In the backward pass, we calculate how much each weight contributed to the overall error and adjust the weights accordingly. The core of this is the chain rule:

$$\frac{\partial E_{total}}{\partial w_{ij}} = \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ij}}$$

This breaks down into the following steps: **(a) Error Calculation:** The error for each output neuron is computed, often using Mean Squared Error (MSE). **(b) Total Error:** The total error is the sum of individual errors: $E_{total} = \sum_j E_j$ **(c) Gradient Calculation:** The chain rule is used to find the gradient of the total error with respect to each weight w_{ij}. Each term is computed as follows:

- $\frac{\partial E_j}{\partial y_j} = -\left(t_j - y_j\right)$ (Derivative of the error w.r.t the output)
- $\frac{\partial y_j}{\partial net_j} = f'\left(net_j\right)$ (Derivative of the activation function)
- $\frac{\partial net_j}{\partial w_{ij}} = x_i$ (Derivative of the net input w.r.t the weight)

(d) Weight Update: The weights are updated using the gradient descent algorithm:

$$w_{ij}^{new} = w_{ij} - \eta \cdot \frac{\partial E_{total}}{\partial w_{ij}}$$

where η is the learning rate.

5.4 Error Calculation

The total error is calculated using the Mean Squared Error (MSE) function:

$$E_{total} = \frac{1}{2}\sum_{j}\left(t_j - y_j\right)^2$$

where t_j is the target output and y_j is the predicted output. This error guides the weight update in the backward pass.

5.5 Activation Functions

Activation functions introduce non-linearity into the model. Common types:

5.5.1 Sigmoid:

Squashes input into (0, 1). Useful for probabilities.

$$f(x) = \frac{1}{1+e^{-x}}$$

Tanh:

Squashes input into (-1, 1). Centered at zero.

$$f(x) = tanh(x)$$

ReLU:

Replaces negative values with zero.

$$f(x) = max(0, x)$$

Leaky ReLU:

Allows a small gradient for negative values.

$$f(x) = \begin{cases} x & x \geq 0 \\ \alpha x & x < 0 \end{cases}$$

ELU:

Exponential Linear Unit, smoothes ReLU's zero gradient issue.

Softmax (Required Addition):

Used in the output layer for **multi-class classification**. It squashes a vector of real numbers into a probability distribution, where all output values are between 0 and 1 and sum to 1.

$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

---

6. Convolutional Neural Networks (CNN)

CNNs are a class of deep neural networks highly effective for analyzing visual imagery.

6.1 Conceptual Advantages

CNNs are effective for images because they **preserve spatial patterns**. A standard network flattens an image, losing all spatial information. A CNN uses filters to scan the image and learn features (like edges, corners, textures) in a way that respects their 2D location.

6.2 The Convolution Operation

A **kernel** (or filter), which is a small matrix of weights, slides (or *convolves*) over the input image. At each position, it performs element-wise multiplication with the part of the image it's on, and sums the result into a single pixel in the output **feature map**. This process detects specific features across the entire image.

6.3 Pooling Layers

- **Purpose:** Pooling layers (e.g., Max Pooling) are used to **reduce the spatial dimensions** (height and width) of the feature map. This reduces the number of parameters, helps control overfitting, and makes detected features more robust to their exact location.
- **Mechanics (Max Pooling):** A window (e.g., 2x2) slides over the feature map. At each step, it outputs only the **maximum** value from that window, discarding the rest.

6.4 Output Size Calculation

The spatial size of the output feature map is:

$$O = \frac{W - K + 2P}{S} + 1$$

- **O**: **Output Size** - The height or width of the feature map generated by the layer.
- **W**: **Input Size** - The height or width of the input volume.
- **K**: **Kernel Size** - The height or width of the convolutional filter (kernel).

- **P**: **Padding** - The number of pixels added to each side of the input. `2P` is used because padding is typically added to both sides (e.g., top and bottom, or left and right).
- **S**: **Stride** - The number of pixels the filter moves at a time as it slides across the input.

6.5 Common Architectures

- **AlexNet:** One of the first deep CNNs to achieve major success (ImageNet 2012). It used stacked convolutional layers, ReLU activation, and dropout for regularization.
- **VGGNet:** Known for its simplicity. It uses *only* small 3x3 convolutional filters stacked in layers of increasing depth, demonstrating that a very deep network can achieve excellent performance.

6.6 Keras/Deep Learning Library Syntax

Libraries like Keras (part of TensorFlow) make building neural networks simple.

- `Sequential`: The most common model, a linear stack of layers.
- `Dense`: A standard, fully-connected neural network layer.
- `input_shape`: Required for the first layer to know what shape of data to expect.

Example: A Simple Classifier

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# 1. Initialize the model
model = Sequential()

# 2. Add layers
# 'input_shape' is for a 784-pixel flattened image (e.g., MNIST)
model.add(Dense(units=64, activation='relu', input_shape=(784,)))
model.add(Dense(units=10, activation='softmax')) # Output layer for 10 classes

# 3. Compile the model (set optimizer, loss, and metrics)
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

model.summary()
```

---

7. Support Vector Machine (SVM)

7.1 Process

(a) **Define Decision Boundary:** Find the hyperplane that best separates classes.

(b) **Maximize Margin:** Ensure the largest distance (margin) between the hyperplane and the nearest data points (support vectors).
(c) **Kernel Trick (if required):** Map data to a higher dimension if it is not linearly separable.
(d) **Solve Optimization Problem:** Use quadratic programming to find optimal hyperplane parameters.

7.2 Formula

$$f(x) = sign(w \cdot x + b)$$

- **w:** Weight vector, defining the orientation of the hyperplane.
- **x:** Input vector (data point).
- **b:** Bias term, shifts the hyperplane.
- **sign():** The function that determines the class label (+1 or -1).

7.3 Important Properties

- Works well for both linear and non-linear data (with kernel functions).
- Robust to outliers using the soft margin.
- Effective in high-dimensional spaces.
- Relatively memory efficient (uses only support vectors for the decision boundary).

7.4 The Kernel Trick

For data that is not linearly separable in its current dimension, SVMs can use the **kernel trick**. A kernel function is a computationally efficient method to transform data into a higher dimension where a linear separator (hyperplane) *can* be found, without having to explicitly calculate the new coordinates.

- **Linear Kernel:** Used for linearly separable data. It is the dot product of the inputs.
- **Polynomial Kernel:** Can find more complex, curved decision boundaries.
- **RBF (Radial Basis Function) Kernel:** A very common and powerful default kernel, effective for most complex, non-linear cases.

---

8. Supervised Learning - Evaluation Metrics

8.1 Confusion Matrix

A Confusion Matrix is a table used to evaluate the performance of a classification model by summarizing the prediction results.

	Predicted: Positive	**Predicted: Negative**
Actual: Positive	True Positive (TP)	False Negative (FN)
Actual: Negative	False Positive (FP)	True Negative (TN)

- **TP (True Positive):** The model correctly predicted the positive class.

- **FP (False Positive):** The model incorrectly predicted the positive class (a "Type I" error).
- **FN (False Negative):** The model incorrectly predicted the negative class (a "Type II" error).
- **TN (True Negative):** The model correctly predicted the negative class.

8.2 Accuracy

The proportion of total predictions that were correct. It is a good general metric but can be misleading for imbalanced datasets.

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$$

8.3 Precision

Answers the question: "Of all the predictions for the positive class, how many were actually correct?"

$$Precision = \frac{TP}{TP+FP}$$

8.4 Recall (Sensitivity)

Answers the question: "Of all the actual positive instances, how many did the model correctly identify?"

$$Recall = \frac{TP}{TP+FN}$$

8.5 F1 Score

The harmonic mean of Precision and Recall. It provides a single score that balances both metrics, and is particularly useful when you have an imbalanced dataset.

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision+Recall}$$

8.6 Mean Absolute Error (MAE) (For Regression)

The average of the absolute differences between the predicted values and the actual values. It measures the average magnitude of the errors in a set of predictions, without considering their direction.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}\left|y_i - \hat{y}_i\right|$$

- y_i: The actual value.
- $\hat{y}_i$: The predicted value.
- n: The total number of samples.

8.7 Mean Squared Error (MSE) (For Regression)

The average of the squared differences between the predicted values and the actual values. It penalizes larger errors more than MAE due to the squaring term.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{y}_i\right)^2$$

- y_i: The actual value.
- $\hat{y}_i$: The predicted value.
- n: The total number of samples.

---

9. Evaluation Metrics for Unsupervised Learning

9.1 Silhouette Score

- **Process:**
  - For each data point i, calculate:
    - $a(i)$: The average distance to all other points in the same cluster (intra-cluster distance).
    - $b(i)$: The average distance to all points in the nearest different cluster (inter-cluster distance).
  - Compute the silhouette score for the point:

$$s(i) = \frac{b(i)-a(i)}{max(a(i),b(i))}$$

  - The overall silhouette score is the mean of all individual scores.

- **Formula:**

$$s(i) = \frac{b(i)-a(i)}{max(a(i),b(i))}$$

  - $s(i)$: Silhouette score for point i, ranging from -1 to 1.
  - $a(i)$: Average distance to points in the same cluster.
  - $b(i)$: Average distance to points in the nearest different cluster.

9.2 Reconstruction Error (Dimensionality Reduction)

- **Process:**
  (a) Reduce the dimensionality of the data using a method like PCA.
  (b) Reconstruct the original data from the reduced dimensions.
  (c) Calculate the error as the difference between the original and reconstructed data.
- **Formula:**

$$\text{Reconstruction Error} = ||X - \hat{X}||^2$$

- X: Original data matrix.
- $\hat{X}$: Reconstructed data matrix from reduced dimensions.
- $||\cdot||^2$: The squared norm (e.g., Euclidean distance) measuring the difference.

9.3 Important Properties

- **Silhouette Score:**
  - Ranges from -1 to 1.
  - Positive scores (closer to 1) indicate dense, well-separated clusters.
  - Scores near 0 suggest overlapping clusters.
  - Negative scores suggest points may have been assigned to the wrong cluster.
- **Reconstruction Error:**
  - Lower values indicate better preservation of the original data in the reduced dimensions.
  - Sensitive to noise and the choice of dimensionality reduction technique.

---

10. K-Means Clustering

10.1 Process

- **Initialize Centroids:** Randomly select k centroids from the dataset.
- **Assign Clusters:** Assign each data point to the nearest centroid (commonly using Euclidean distance).

- **Update Centroids:** Recalculate the centroids as the mean of all points assigned to that cluster.
- **Repeat:** Iterate steps 2 and 3 until the centroids stabilize or a maximum number of iterations is reached.

10.2 Formula

K-Means involves two key calculations: updating the centroid and measuring the distance to it.

Centroid Calculation

The centroid μ_j for a cluster C_j is calculated as the mean of all data points x_i belonging to that cluster.

$$\mu_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i$$

- Where $|C_j|$ is the number of points in cluster j.

Distance Metric

K-Means most commonly uses **Euclidean distance** to assign a point to the nearest centroid. The formula to calculate the distance between a data point x and a centroid μ_j is:

$$d(x, \mu_j) = \sqrt{\sum_{i=1}^{d} (x_i - \mu_{ji})^2}$$

- Where d is the number of dimensions (features), x_i is the i-th feature of the data point, and μ_{ji} is the i-th feature of the centroid. This calculates the standard straight-line distance between the two points.

10.3 Important Properties

- **Unsupervised Learning:** No labeled data is required.
- **Partitioning Algorithm:** Divides data into k distinct, non-overlapping clusters.
- **Centroid-Based:** Aims to minimize within-cluster variance (the sum of squared distances between points and their centroid).
- **Sensitive to Initialization:** The final clusters can vary depending on the initial random placement of centroids.
- **Scalability:** Efficient for large datasets but can be slow with a very high number of dimensions or clusters.

10.4 The Elbow Method

The Elbow Method is a popular technique used to find the optimal number of clusters (k) for a dataset.

1. Run the K-Means algorithm for a range of k values (e.g., from 1 to 10).
2. For each value of k, calculate the **WCSS (Within-Cluster Sum of Squares)**. This is the sum of the squared distances between each point and its centroid within a cluster.
3. Plot the WCSS values for each k.
4. The "elbow point" on the plot—the point where the rate of decrease in WCSS sharply slows down—is considered the optimal number of clusters.

---

11. Hierarchical Clustering

11.1 Agglomerative (Bottom-Up) Process

- Starts by treating each data point as a single, individual cluster.
- Repeatedly merges the two closest clusters based on a linkage criterion.
- This process continues until all data points are in a single, large cluster.
- The result is a **dendrogram**, a tree-like diagram that shows the hierarchy of merges.

11.2 Linkage Methods

This determines how the distance between clusters is measured:

- **Single Linkage:** The distance is the shortest distance between any two points in the two different clusters.
- **Complete Linkage:** The distance is the longest distance between any two points in the two different clusters.
- **Average Linkage:** The distance is the average distance between all pairs of points in the two clusters.
- **Ward's Method:** Merges the two clusters that result in the minimum increase in the total within-cluster variance.

---

12. Density-Based Spatial Clustering of Applications with Noise (DBSCAN)

12.1 Key Concepts

- **Epsilon (ϵ):** A distance that defines a point's "neighborhood."
- **MinPts:** The minimum number of points required to form a dense region.
- **Core Point:** A point with at least MinPts in its ϵ-neighborhood.
- **Noise Point:** A point that is not a Core Point and is not reachable by any Core Point.

12.2 Process

1. Start with a random point. If it is a core point, a new cluster is formed.
2. Expand the cluster by adding all density-reachable points.
3. Repeat until all points have been visited. Points not assigned to any cluster are noise.

---

13. Computer Vision

13.1 Key Tasks

- **Image Classification:** Assigns a single label to an image.
- **Object Detection:** Locates and identifies objects within an image using bounding boxes.
- **Image Segmentation:** Partitions an image into regions based on object boundaries.

13.2 Common Architectures

13.2.1 LeNet-5

- One of the first Convolutional Neural Networks (CNNs).
- Used for handwritten digit recognition.
- **Structure:** Alternating convolutional and pooling layers, ending with fully connected layers.

13.2.2 AlexNet

- Won the 2012 ImageNet competition.
- A deeper network that used ReLU to improve training speed.
- Employed Dropout to prevent overfitting.
- Utilized Data Augmentation to increase data diversity.

13.2.3 VGG16

- A deep network with a simple and uniform structure.
- Consists of 16 layers with weights.
- Uses a stack of small 3x3 convolutional filters.

13.2.4 GoogLeNet (Inception)

- Won the 2014 ImageNet competition.
- Key component is the Inception module, which performs parallel convolutions with different filter sizes.
- Uses 1x1 convolutions as a bottleneck layer to reduce computation.
- Replaced fully connected layers with Global Average Pooling to reduce parameters.

13.2.5 Residual Networks (ResNet)

- Won the 2015 ImageNet competition.
- Solves the vanishing gradient problem in very deep networks.
- The main innovation is the residual block.

- Uses skip connections to add input directly to the output of a block.

13.2.6 Densely Connected Convolutional Networks (DenseNet)

- Improves information flow and feature reuse.
- Within a Dense Block, each layer is connected to all preceding layers.
- The input to a layer is a concatenation of feature maps from all previous layers.
- Achieves high parameter efficiency.