Summer Internship Report

Introduction to Machine Learning and Deep Learning

Submitted By:

Naresh Lankalapalli

S200812

Department of Electronics and communication

RGUKT-SKLM

Submitted To:

Prof.J Ravi Kumar

NIT-Warangal

Department of Electronics and communication

Declaration

I Naresh Lankalapalli, hereby declare that the Summer Internship report entitled "Implementing Deep Neural Networks on FPGA" is a record of the original work carried out by me during my internship at National Institute of Technology, Warangal under the guidance of Prof. Ravikumar J.

This work has not been submitted to any other university or institution for the award of any degree, diploma, or certificate. The information presented in this report is based on the results obtained during the internship period and is true to the best of my knowledge and belief.

Place: Warangal
Date: 10-07-2025

Naresh Lankalapalli
Reg No:S200812

Certificate

This is to certify that Mr./Ms. Naresh Lankalapalli, a student of RGUKTSKLM, has successfully completed his Summer Internship at National Institute of Technology, Warangal from 10-05-2025 to 10-07-2025 under my supervision.

The internship work entitled "Implementing Deep Neural Networks on FPGA" has been carried out by the student under my guidance and is a bonafide record of the work performed.

I certify that the work reported herein is original and has been carried out by the student to the best of his ability.

Prof. Ravikumar J
(Guide & Supervisor)
Department of ECE
National Institute of Technology, Warangal

Acknowledgment

I express my sincere gratitude to Prof. Ravikumar J, my internship guide at National Institute of Technology, Warangal, for his constant guidance, encouragement, and support throughout the course of my internship. His invaluable suggestions and expertise in the field of FPGA-based deep learning greatly contributed to the successful completion of my work.

I would also like to thank the Department of ECE at NIT Warangal for providing me with the necessary facilities and a conducive environment to carry out my project.

My heartfelt thanks to my faculty members, friends, and family for their constant support and encouragement during this internship period.

Contents

1. Python Basics

1.1 Python Basics: Cheat Sheet with Examples

Hello World

Command:

```
print("Hello, World!")
```

Output:

```
Hello, World!
```

Variables

Code:

```
age = 18 # age is of type int
name = "cherry" # name is now of type str
print(name)
```

Output:

```
cherry
```

Data Types

Common built-in data types in Python:

- `str` – Text
- `int`, `float`, `complex` – Numeric types
- `list`, `tuple`, `range` – Sequence types
- `dict` – Mapping type
- `set`, `frozenset` – Set types

- `bool` – Boolean type
- `bytes`, `bytearray`, `memoryview` – Binary types

See official docs: Python Data Types

String Slicing

Code:

```
msg = "Hello, World!"
print(msg[2:5])
```

Output:

```
llo
```

Lists

Code:

```
mylist = []
mylist.append(1)
mylist.append(2)
for item in mylist:
    print(item)
```

Output:

```
1
2
```

If-Else

Code:

```
num = 200
if num > 0:
    print("num is greater than 0")
else:
    print("num is not greater than 0")
```

Output:

```
num is greater than 0
```

Loops (For-Else)

Code:

```
for item in range(6):
    if item == 3:
        break
    print(item)
else:
    print("Finally finished!")
```

Output:

```
0
1
2
```

Explanation: The `else` block is skipped when `break` is triggered.

Functions

Code:

```
def my_function():
    print("Hello from a function")

my_function()
```

Output:

```
Hello from a function
```

Explanation: Functions in Python are defined using `def` keyword.

Arithmetic Operators

Code:

```
result = 10 + 30 # => 40
result = 40 - 10 # => 30
result = 50 * 5 # => 250
result = 16 / 4 # => 4.0 (Float Division)
result = 16 // 4 # => 4 (Integer Division)
result = 25 % 2 # => 1
result = 5 ** 3 # => 125
```

Explanation:

- `/` – Float division

- `//` – Integer (floor) division
- `%` – Modulus
- `**` – Power

f-Strings (Python 3.6+)

Code:

```
website = 'RGUKT_SKLM'
print(f"Hello, {website}")

num = 10
print(f'{num} + 10 = {num + 10}')
```

Output:

```
Hello,RGUKTSKLM
10 + 10 = 20
```

Explanation: f-Strings offer inline expressions and improved readability.

Working with Strings

Single-Line Strings:

```
greet1 = "Hello World"
greet2 = 'Hello World'
```

Multi-Line String:

```
multi_text = """Multiline Strings
Lorem ipsum dolor sit amet,
consectetur adipiscing elit"""
```

Explanation: Triple quotes (`""" """`) are used for multi-line strings. *if it was not assigned to any variable its act like multiline comment*

Understanding Number Types

```
x = 1 # Integer
y = 2.8 # Float
z = 1j # Complex

print(type(x)) # Output: <class 'int'>
```

Explanation: Python supports `int`, `float`, and `complex` number types natively.

Booleans

```
my_bool = True
my_bool = False

print(bool(0)) # False
print(bool(1)) # True
```

Explanation: The `bool()` function evaluates the truthiness of a value.

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

print(person["name"]) # Output: Alice
print(person.keys()) # dict_keys(['name', 'age', 'city'])
print(person.values()) # dict_values(['Alice', 25, 'Hyderabad'])
person.update({"email": "alice@example.com"})
print(person["email"]) # Output: alice@example.com
```

Explanation: Dictionaries are like JSON objects — ideal for storing structured data.

Type Casting (Conversions)

To Integer:

```
a = int(1) # 1
b = int(2.8) # 2
c = int("3") # 3
```

To Float:

```
a = float(1) # 1.0
b = float("3.5") # 3.5
```

To String:

```
a = str("Python") # 'Python'
b = str(2) # '2'
c = str(3.14) # '3.14'
```

Explanation: Python supports type casting between basic types for flexible usage.

2. Python libraries for Machine Learning

2.1 NumPy Arrays and Operations

Importing NumPy:

```
import numpy as np
```

Creating Arrays:

```
a = np.array([1, 2, 3])
b = np.array([[1, 2], [3, 4]])
```

Array Attributes:

```
print(a.shape) # (3,)
print(b.shape) # (2, 2)
print(a.dtype) # int64 (depends on your system)
print(b.ndim) # 2
```

Array Indexing and Slicing:

```
a = np.array([10, 20, 30, 40])
print(a[1]) # 20
print(a[1:3]) # [20 30]

b = np.array([[1, 2, 3], [4, 5, 6]])
print(b[1, 2]) # 6
print(b[:, 1]) # [2 5]
```

Array Operations:

```
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])

print(x + y) # [5 7 9]
print(x * y) # [ 4 10 18]
print(x ** 2) # [1 4 9]
print(np.dot(x, y)) # 32
```

Reshaping Arrays:

```
c = np.array([[1, 2, 3], [4, 5, 6]])
reshaped = c.reshape(3, 2)
print(reshaped)
# [[1 2]
# [3 4]
# [5 6]]
```

Useful Functions:

```
np.arange(0, 10, 2) # [0 2 4 6 8]
np.linspace(0, 1, 5) # [0. 0.25 0.5 0.75 1. ]
```

Aggregation Functions:

```
a = np.array([1, 2, 3, 4])

print(np.min(a)) # 1
print(np.max(a)) # 4
print(np.sum(a)) # 10
print(np.mean(a)) # 2.5
print(np.std(a)) # 1.118...
```

Boolean Masking:

```
a = np.array([1, 2, 3, 4, 5])
mask = a > 3
print(a[mask]) # [4 5]
```

Matrix Operations in NumPy

Matrix Multiplication:

```
import numpy as np

A = np.array([[1, 2], [3, 4]])
B = np.array([[2, 0], [1, 2]])

# Matrix product
C = np.dot(A, B)
print(C)
# [[4 4]
# [10 8]]

# Shortcut using @ operator (Python 3.5+)
print(A @ B)
```

Transpose and Inverse:

```
# Transpose
print(A.T)

# Inverse (only if matrix is square and non-singular)
inv_A = np.linalg.inv(A)
print(inv_A)

# Determinant
det = np.linalg.det(A)
print(det)
```

Eigenvalues and Eigenvectors:

```
eigenvalues, eigenvectors = np.linalg.eig(A)
print("Eigenvalues:", eigenvalues)
print("Eigenvectors:", eigenvectors)
```

Broadcasting in NumPy

Basic Broadcasting Example:

```
a = np.array([1, 2, 3])
b = 2
print(a + b) # [3 4 5]
```

Broadcasting with Arrays:

```
a = np.array([[1, 2, 3],
              [4, 5, 6]])

b = np.array([1, 0, 1])

print(a + b)
# [[2 2 4]
# [5 5 7]]
```

Shape Rules:

- If two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading side.
- If the shape of the two arrays in a dimension is equal or one of them is 1, the arrays are compatible.

2.2 Pandas Basics and Examples

Importing Pandas:

```
import pandas as pd
```

Creating a Series:

```
data = [10, 20, 30, 40]
s = pd.Series(data)
print(s)
# 0 10
# 1 20
# 2 30
# 3 40
# dtype: int64
```

Creating a DataFrame:

```
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [25, 30, 35, 40],
    'City': ['New York', 'Paris', 'London', 'Tokyo']
}

df = pd.DataFrame(data)
print(df)
```

Selecting Columns:

```
print(df['Name'])
```

Selecting Rows by Index:

```
print(df.loc[1]) # Row with index 1 (Bob)
print(df.iloc[2]) # 3rd row (Charlie)
```

Filtering Rows:

```
print(df[df['Age'] > 30])
```

Adding a New Column:

```
df['Senior'] = df['Age'] > 30
print(df)
```

Basic Statistics:

```
print(df.describe())
```

Reading and Writing CSV Files:

```
df.to_csv('people.csv', index=False)
df2 = pd.read_csv('people.csv')
print(df2)
```

2.3 Matplotlib.pyplot Basics and Examples

Importing Matplotlib:

```
import matplotlib.pyplot as plt
```

Basic Line Plot:

```
x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 7, 11]

plt.plot(x, y)
plt.title("Basic Line Plot")
plt.xlabel("X axis")
plt.ylabel("Y axis")
plt.show()
```

Plotting Multiple Lines:

```
x = range(1, 6)
y1 = [1, 4, 9, 16, 25]
y2 = [1, 8, 27, 64, 125]

plt.plot(x, y1, label="Squares")
plt.plot(x, y2, label="Cubes", linestyle='--')
plt.title("Multiple Lines")
plt.xlabel("X axis")
plt.ylabel("Y axis")
plt.legend()
plt.show()
```

Scatter Plot:

```
import numpy as np

x = np.random.rand(50)
y = np.random.rand(50)
sizes = 100 * np.random.rand(50)
colors = np.random.rand(50)

plt.scatter(x, y, s=sizes, c=colors, alpha=0.5)
plt.title("Scatter Plot with Size and Color")
plt.show()
```

Bar Chart:

```
labels = ['A', 'B', 'C', 'D']
values = [5, 7, 3, 4]

plt.bar(labels, values, color='skyblue')
```

```
plt.title("Bar Chart")
plt.show()
```

Histogram:

```
data = np.random.randn(1000)

plt.hist(data, bins=30, color='green', alpha=0.7)
plt.title("Histogram")
plt.show()
```

Customizing Plot Appearance:

```
plt.plot(x, y1, color='red', linewidth=2, marker='o', markersize=8)
plt.title("Customized Plot")
plt.grid(True)
plt.show()
```

Saving Plot as Image:

```
plt.plot(x, y1)
plt.savefig('myplot.png') # Saves the current figure
plt.close() # Closes the figure window
```

Pie Chart using matplotlib.pyplot

Basic Pie Chart:

```
import matplotlib.pyplot as plt

labels = ['Apples', 'Bananas', 'Cherries', 'Dates']
sizes = [30, 25, 25, 20]

plt.pie(sizes, labels=labels)
plt.title("Basic Pie Chart")
plt.show()
```

Pie Chart with Explode and Percentage:

```
explode = (0.1, 0, 0, 0) # 'Explode' first slice slightly

plt.pie(sizes, labels=labels, explode=explode, autopct='%1.1f%%',
        shadow=True, startangle=140)
plt.title("Pie Chart with Explode and Percentages")
plt.show()
```

Pie Chart with Custom Colors:

```
colors = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue']
```

```
plt.pie(sizes, labels=labels, colors=colors, autopct='%1.1f%%',
        startangle=90)
plt.title("Pie Chart with Custom Colors")
plt.axis('equal') # Equal aspect ratio ensures pie is circular.
plt.show()
```

2.4 SciPy Basics and Examples

Importing SciPy modules:

```
from scipy import optimize, integrate, stats
```

1. Root Finding using `optimize`:

```
# Define a function
def f(x):
    return x**3 - 1

# Find root near x=1
root = optimize.root_scalar(f, bracket=[0, 2], method='brentq')
print("Root of f(x) = 0 is:", root.root)
```

2. Numerical Integration using `integrate`:

```
# Integrate function x^2 from 0 to 3
result, error = integrate.quad(lambda x: x**2, 0, 3)
print("Integral of x^2 from 0 to 3 is:", result)
```

3. Statistical Functions using `stats`:

```
data = [1, 2, 2, 3, 4, 5, 5, 5, 6]

mean = stats.tmean(data)
median = stats.scoreatpercentile(data, 50)
mode = stats.mode(data)
std_dev = stats.tstd(data)

print(f"Mean: {mean}")
print(f"Median: {median}")
print(f"Mode: {mode.mode[0]}, Count: {mode.count[0]}")
print(f"Standard Deviation: {std_dev}")
```

4. Optimization Example: Minimizing a Function

```
def func_to_minimize(x):
    return (x - 3)**2 + 4

result = optimize.minimize(func_to_minimize, x0=0)
```

```
print("Minimum value found at x =", result.x)
```

2.5 Python os Library Basics

Importing the os Module:

```
import os
```

1. Get Current Working Directory:

```
cwd = os.getcwd()
print("Current Working Directory:", cwd)
```

2. List Files and Directories:

```
entries = os.listdir(cwd)
print("Entries in directory:", entries)
```

3. Create a New Directory:

```
os.mkdir('new_folder')
```

4. Rename a File or Directory:

```
os.rename('old_name.txt', 'new_name.txt')
```

5. Remove a File:

```
os.remove('file_to_delete.txt')
```

6. Check if a Path Exists:

```
exists = os.path.exists('some_path')
print("Path exists:", exists)
```

7. Join Paths:

```
full_path = os.path.join(cwd, 'new_folder', 'file.txt')
print(full_path)
```

8. Environment Variables:

```
home = os.environ.get('HOME') # or 'USERPROFILE' on Windows
print("Home directory:", home)
```

3. World of Machine Learning

3.1 Introduction to Machine Learning

3.1.1 What is Machine Learning?

Imagine you're teaching your little sibling how to recognize fruits.

You show them a red, round fruit and say, "This is an apple." Then you show them a long, yellow fruit and say, "This is a banana." After a few tries, your sibling starts guessing correctly on their own.

That's **Machine Learning** in its simplest form.

3.1.2 Why is it Important?

Today, ML is everywhere:

- **Netflix:** Suggests movies based on what you've watched.
- **Google Maps:** Predicts the fastest route using traffic data.
- **Instagram:** Filters spam comments and recommends posts.
- **Finance Apps:** Detects unusual credit card activity.

We live in a world filled with data — and Machine Learning helps make sense of it, find patterns, and take smart actions.

3.1.3 A Real-Life example

"Teaching a machine is like training a dog."

- You give examples — say "sit" and reward when it does right (Training data).
- Over time, it learns to sit when you say the word (Model learns from data).
- If it sees something new — like a new gesture — and still sits, it has generalized well (Testing).

The more consistent and clear the examples, the better the dog (or machine) learns!

3.1.4 Key Idea

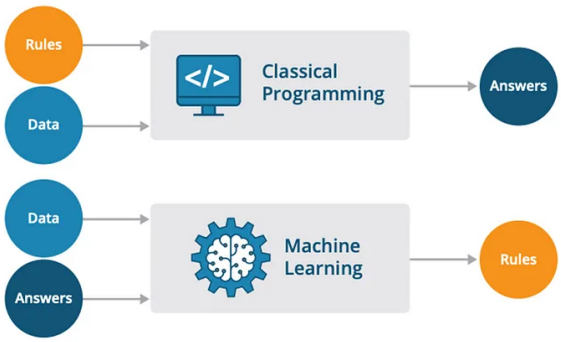


Figure 3.1: Traditional vs ML

Traditional programming: Rules + Data → Output
Machine Learning: Data + Output → *Learns the Rules*

That means we don't tell the computer how to solve the problem — we show it examples, and it figures out the rules.

Machine Learning means teaching computers to learn from data, so they can make decisions or predictions without being told exactly what to do

"Give the computer data and let it figure it out."

3.2 Machine Learning Types

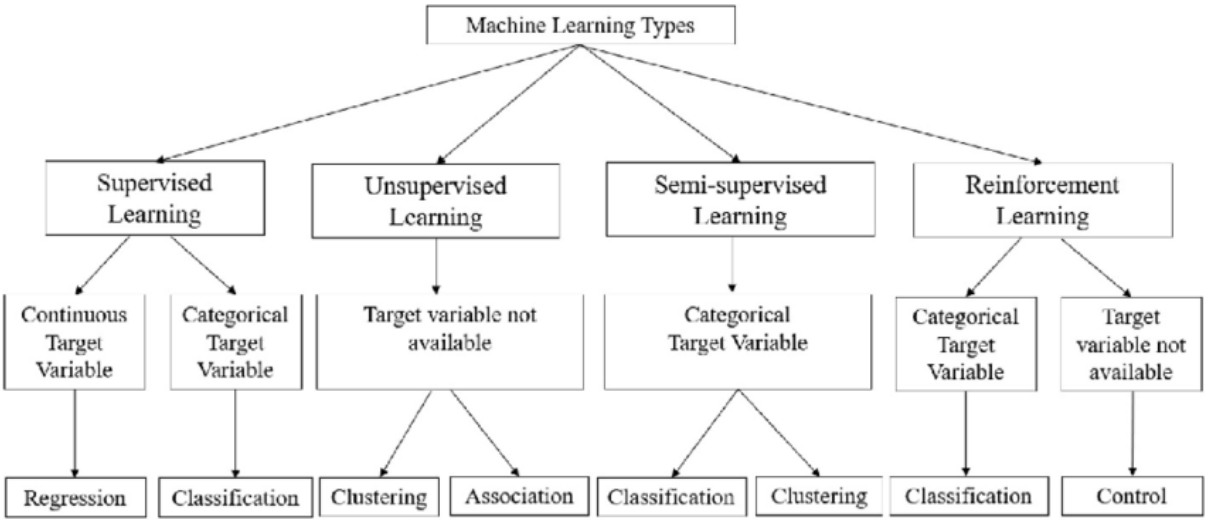


Figure 3.2: Machine learning

3.2.1 Supervised Learning

Think a Minute

How does your email app know if a message is spam or not?

Definition: Machine Learning where the model learns from labeled data — it knows the correct answers during training.

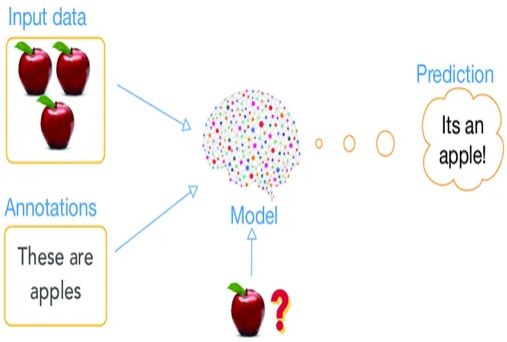


Figure 3.3: Supervised Learning

3.2.2 Unsupervised Learning

Think a Minute

Have you noticed how Google groups your photos by people or places automatically?

Definition: Machine Learning where the model finds patterns or groups in data without any labeled answers.

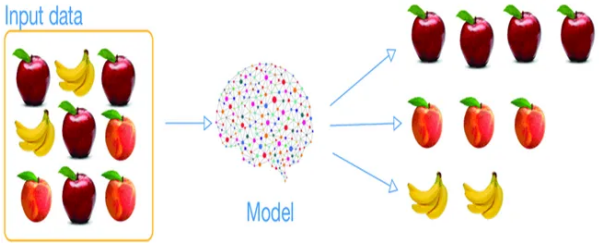


Figure 3.4: Unsupervised Learning

3.2.3 Reinforcement Learning

Think a Minute

Imagine a video game character learning to win by trying different moves and getting points or losing lives.

Definition: Machine Learning where an agent learns by trial and error, receiving rewards or penalties based on actions.

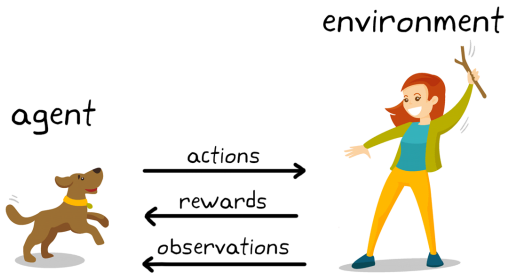


Figure 3.5: Reinforcement Learning

3.2.4 Classification vs Regression

In Supervised Learning, problems are mainly of two types: **Classification** and **Regression**.

Classification:

Think a Minute – Classification

Ever tried sorting candies into colors? Red ones here, green ones there, yellow ones over there.

This is about teaching a machine to *choose a category*. It answers questions like: "*Which group does this belong to?*"

Example: Your phone unlocks only when it sees your face. It's deciding: `Is this YOU or NOT YOU?`

Regression:

Think a Minute – Regression

Have you ever guessed how much a full tank of fuel will cost based on how empty your tank is? That's regression — predicting a value based on other info!

This is about teaching a machine to *predict a number*. It answers questions like: "*How much? or What value?*"

Example: Your food delivery app estimates how long it will take for your pizza to arrive — that's predicting a number (in minutes).

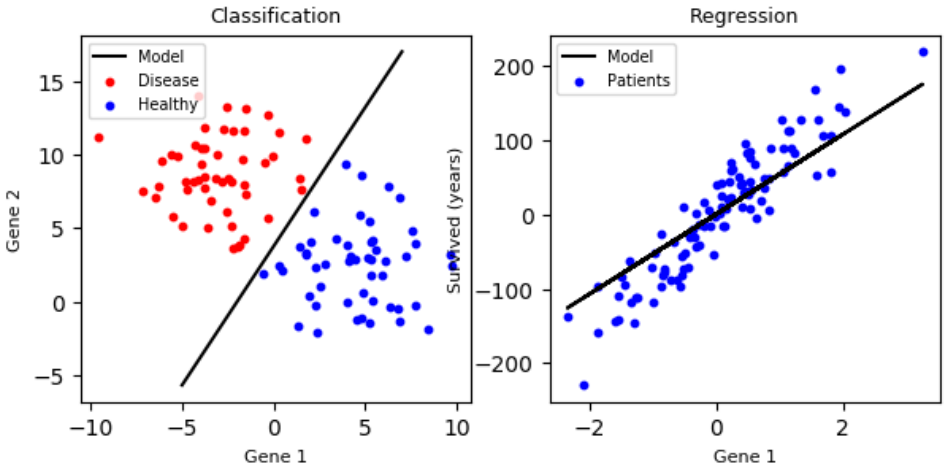


Figure 3.6: Classification vs regression

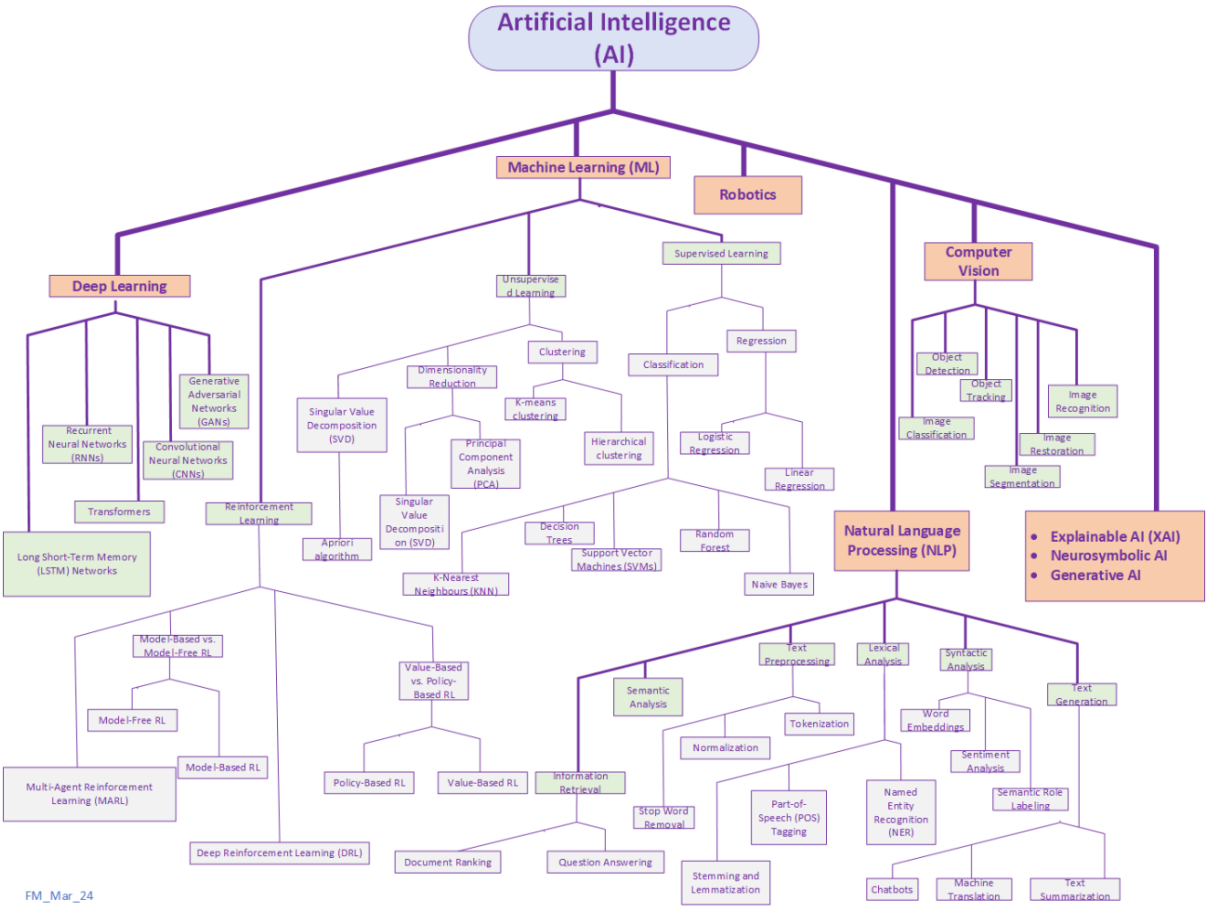


Figure 3.7: AI-Tree

3.2.5 Random Data

Imagine you have some data — for example, the heights of students in a class, or the scores of a game. This data is just a list of numbers.

What is a Parameter?

A **parameter** is just a number that helps describe your data in a simple way.

Two very important parameters are:

- **Mean** (Average) — tells you the center or typical value of the data.
- **Variance** — tells you how spread out or varied the data is.

Why do we care about Mean and Variance?

Mean: The Average Value The mean is like the "balance point" of all your data points.

Variance: How Spread Out Is the Data? Variance measures how much the numbers differ from the mean.

$$\text{Variance} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$$

where x_i are data points, μ is the mean, and N is the number of points.

- If variance is small, data points are close to the mean.
- If variance is large, data points are more spread out.

How Does This Help in Machine Learning?

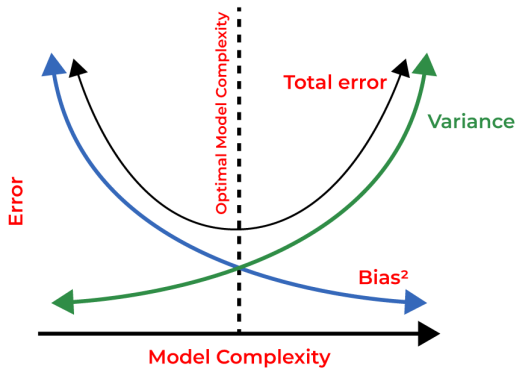
Think a Minute

Why is it useful to know the average and spread of your data before training a model?

Hint: Think about what would happen if the data has wildly different scales or if some points are very far from the rest.

Machine learning models need to **understand** the data to make good predictions. Mean and variance help in:

- **Summarizing the data:** Quickly get a sense of where most data points lie.



- **Normalizing data:** Making data easier to work with by adjusting it relative to its mean and variance.
- **Detecting outliers:** Points far from the mean might be errors or special cases.
- **Improving model performance:** Many algorithms assume data is centered and scaled properly.

Think a Minute

If two datasets have the same mean but very different variance, how might they look different? Which one might be easier for a model to learn from?

4. Supervised Learning

4.1 From Linear Models to Linear Regression

A Linear Probabilistic Model — What's That?

Imagine you're trying to predict someone's marks based on the number of hours they study.

The relationship might look like this:

$$\text{Marks} = a \cdot (\text{Hours Studied}) + b + \text{Some Random Noise}$$

This is called a **linear probabilistic model**: - It assumes a *linear relationship* between the input (hours) and output (marks). - But in real life, nothing is perfect — so we add random noise to make the model realistic.

Think a Minute

If two students study the same number of hours, will they always score exactly the same? Of course not — that's why we need to model the randomness!

What Are We Trying to Find?

We want to find the best line that "fits" the data — in our example, it's the line that predicts marks based on hours.

That line has two parameters:

- a — the **slope**, which tells how much marks increase for each extra hour studied.
- b — the **intercept**, which is the baseline marks even if no hours were studied.

Real Life Analogy: Drawing the Best Line

Imagine plotting points for students' marks vs hours studied. You try placing a ruler such that it touches as many points as possible — that's what the model is doing mathematically!

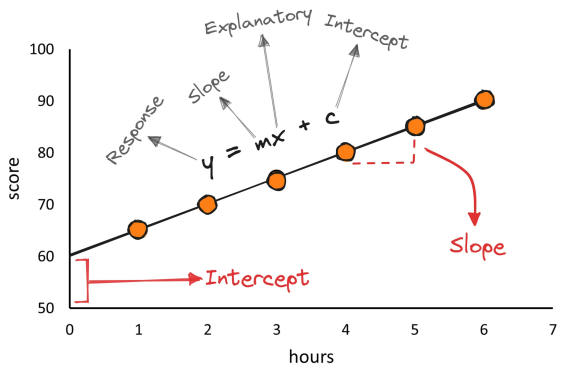


Figure 4.1: Simple linear relation

How Does This Become Machine Learning?

When we apply this idea in machine learning:

- Our inputs (x) can be many features — not just hours studied.
- We still want to learn the best parameters (weights) $w_1, w_2, ..., w_n$ and bias b.
- We use the same idea of minimizing errors to train the model.

$$\text{Prediction} = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$$

This is called a **linear regression model** in machine learning.

4.2 What is Linear Regression?

The model assumes that the relationship between the input (x) and the output (y) is linear — that means it can be drawn as a straight line.

$$\text{Prediction } (\hat{y}) = w \cdot x + b$$

we have to find w and b
Where:

- w is the **slope** (how much y changes when x increases),
- b is the **intercept** (value of y when $x = 0$),
- x is the input (feature),
- $\hat{y}$ is the predicted output.

Where Do We Use It?

- Predicting house prices based on area
- Estimating sales based on advertising budget
- Predicting student performance from study time

Think a Minute

How Do We Know If Our Model Is Good?

After making predictions, we need to check how good they are . We do this using evaluation metrics.

1. Mean Absolute Error (MAE)

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

It tells us: "On average, how far off are we?"

- Easy to understand
- Treats all errors equally

2. Mean Squared Error (MSE)

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

It penalizes bigger errors more than smaller ones (because of squaring).

3. Root Mean Squared Error (RMSE)

$$\text{RMSE} = \sqrt{\text{MSE}}$$

This is in the same units as the output, making it easier to interpret.

4. R-Squared (R^2 Score)

$$R^2 = 1 - \frac{\text{SS}_{\text{res}}}{\text{SS}_{\text{tot}}}$$

Where:

$$\text{SS}_{\text{res}} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \quad \text{(Residual Sum of Squares)}$$

$$\text{SS}_{\text{tot}} = \sum_{i=1}^{n}(y_i - \bar{y})^2 \quad \text{(Total Sum of Squares)}$$

This tells us how much of the variation in the data is explained by the model.

- $R^2 = 1$ means perfect prediction
- $R^2 = 0$ means model is no better than just guessing the average

Think a Minute

If your predictions are close to the actual values, which metric would show the smallest value? Hint: MAE and RMSE are error values — smaller is better!

4.2.1 Least Squares Method — Best Fit Line from Math

To find this best line, we use the **least squares method**, which means:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - (b_1 x_i + b_0))^2$$

where:

- y_i: actual value,
- $\hat{y}_i = b_1 x_i + b$: predicted value,
- n: number of data points.

The optimal values of b_1 and b_0 are found by minimizing this error function, typically using calculus or optimization techniques like gradient descent.

Think a Minute

Have you ever tried guessing someone's marks based on how much they studied? Congrats — you've already done a form of regression!

4.2.2 Math Behind Linear Regression from Scratch

Objective: We aim to find the best-fit line of the form $y = b_0 + b_1 x$ that minimizes the total error between actual and predicted values.

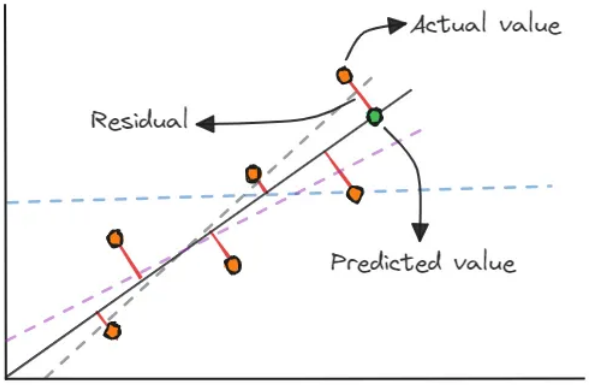


Figure 4.2: Residual vs. actual value vs. predicted value in simple linear regression

Step 1: Residuals

Residual for each point:

$$\varepsilon_i = y_i - (b_0 + b_1 x_i)$$

Sum of squared residuals:

$$S_r = \sum_{i=1}^{n} \varepsilon_i^2 = \sum_{i=1}^{n} (y_i - b_0 - b_1 x_i)^2$$

Step 2: Minimize the Error

To minimize S_r, take partial derivatives with respect to b_0 and b_1 and set them to zero.

With respect to b_0:

$$\frac{\partial S_r}{\partial b_0} = -2 \sum (y_i - b_0 - b_1 x_i)$$

With respect to b_1:

$$\frac{\partial S_r}{\partial b_1} = -2 \sum x_i (y_i - b_0 - b_1 x_i)$$

Step 3: Solve the Equations

From $\frac{\partial S_r}{\partial b_0} = 0$:

$$\sum y_i - n b_0 - b_1 \sum x_i = 0 \Rightarrow b_0 = \bar{y} - b_1 \bar{x}$$

Substitute into the second derivative equation:

$$\sum x_i (y_i - b_0 - b_1 x_i) = 0 \Rightarrow \sum x_i (y_i - \bar{y} + b_1 \bar{x} - b_1 x_i) = 0$$

$$\Rightarrow \sum x_i (y_i - \bar{y}) - b_1 \sum x_i (x_i - \bar{x}) = 0$$

$$\Rightarrow b_1 = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2}$$

Final Formulas

$$b_1 = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2} \quad , \quad b_0 = \bar{y} - b_1\bar{x}$$

Useful Identities

$$\sum(x_i - \bar{x})(y_i - \bar{y}) = \sum x_i(y_i - \bar{y}) \quad \text{because } \sum(y_i - \bar{y}) = 0$$

$$\sum(y_i - \bar{y})^2 = \sum y_i(y_i - \bar{y}) \quad \text{by same reason}$$

Karl Pearson's Coefficient of Correlation

Karl Pearson's coefficient of correlation, denoted by r, measures the **strength and direction of the linear relationship** between two variables .i.e measures the quality of prediction . It is also known as the **Product Moment Correlation Coefficient**.

1. Formula (Using Deviations):

$$r = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum(x_i - \bar{x})^2 \cdot \sum(y_i - \bar{y})^2}}$$

2. Formula (Product Method / Raw Scores):

$$r = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{(n\sum x_i^2 - (\sum x_i)^2)(n\sum y_i^2 - (\sum y_i)^2)}}$$

3. Interpretation of r:

Value of r	Interpretation
$r = +1$	Perfect positive correlation (x and y increase together exactly)
$0.7 \leq r < 1$	Strong positive correlation
$0.3 \leq r < 0.7$	Moderate positive correlation
$0 < r < 0.3$	Weak positive correlation
$r = 0$	No linear correlation
$-0.3 < r < 0$	Weak negative correlation
$-0.7 < r \leq -0.3$	Moderate negative correlation
$-1 < r \leq -0.7$	Strong negative correlation
$r = -1$	Perfect negative correlation (x increases, y decreases exactly)

4. Conditions for Using Pearson's r:

- The data must be **quantitative and continuous**.
- The relationship between the variables should be **linear**.
- There should be **no extreme outliers**.
- Both variables should be approximately **normally distributed**.

Coefficient of Determination R^2

The coefficient of determination R^2 tells how much of the variance in Y is explained by X through the regression:

$$\boxed{R^2 = r^2}$$

This means the proportion of variance explained by the regression model is exactly the square of the correlation coefficient between X and Y.

—

Range:

$$0 \leq R^2 \leq 1$$

where

- $R^2 = 0$ means no variability explained by the model.
- $R^2 = 1$ means perfect fit to the data.

Avoiding Extrapolation

What is Extrapolation?

Extrapolation is the process of making predictions **outside the range** of the observed data used to build a model. For example, if the data for the independent variable X covers values from 10 to 50, then predicting Y for $X = 60$ is extrapolation.

Significance of F-value and P-value in Regression

F-value: Overall Model Significance

The **F-value** is used in an *F-test* to evaluate the overall significance of a linear regression model. It compares the model with no predictors (intercept-only model) against your full model.

$$F = \frac{\text{Explained Variance (Mean Square Regression)}}{\text{Unexplained Variance (Mean Square Error)}}$$

- A high F-value means your model explains a lot more variation than what you'd expect by chance.
- If the F-value is significantly greater than 1, the model is considered meaningful.

Think a Minute

Imagine you're testing if studying time predicts exam scores. A high F-value means studying actually explains the difference in scores across students.

P-value: Feature Significance

The **P-value** tests the null hypothesis that the coefficient (b_i) of a variable is equal to zero (i.e., no effect).

$$\text{Null Hypothesis: } H_0 : b_i = 0 \quad vs \quad \text{Alternative Hypothesis: } H_1 : b_i \neq 0$$

- A **low p-value** (typically ≤ 0.05) indicates that you can reject the null hypothesis — meaning the variable is statistically significant.
- A **high p-value** means the variable is likely not contributing much to the prediction.

Think a Minute

Suppose you're predicting house prices. If the P-value for "number of bathrooms" is low, it's a useful predictor. If it's high, the model might do fine without it.

Example: Linear Regression by Least Square Method

Given:

$$x = (95,\ 85,\ 80,\ 70,\ 60), \quad y = (85,\ 95,\ 70,\ 65,\ 70)$$

First, calculate the means:

$$\bar{x} = \frac{95 + 85 + 80 + 70 + 60}{5} = 78, \quad \bar{y} = \frac{85 + 95 + 70 + 65 + 70}{5} = 77$$

x	**y**	$x - \bar{x}$	$y - \bar{y}$	$(x - \bar{x})(y - \bar{y})$	$(x - \bar{x})^2$	$(y - \bar{y})^2$
95	85	17	8	136	289	64
85	95	7	18	126	49	324
80	70	2	-7	-14	4	49
70	65	-8	-12	96	64	144
60	70	-18	-7	126	324	49
Total				**470**	**730**	**630**

Now calculate the slope b_1 and intercept b_0:

$$b_1 = \frac{\sum(x - \bar{x})(y - \bar{y})}{\sum(x - \bar{x})^2} = \frac{470}{730} \approx 0.644$$

$$b_0 = \bar{y} - b_1\bar{x} = 77 - (0.644 \times 78) \approx 77 - 50.23 = 26.77$$

$$\boxed{y = 26.77 + 0.644x}$$

This is the estimated regression line.

Correlation Coefficient and Coefficient of Determination

Step 3: Pearson Correlation Coefficient

$$r = \frac{\sum(x - \bar{x})(y - \bar{y})}{\sqrt{\sum(x - \bar{x})^2 \cdot \sum(y - \bar{y})^2}} = \frac{470}{\sqrt{730 \cdot 630}}$$

$$r = \frac{470}{\sqrt{459900}} = \frac{470}{678.22} \approx 0.693$$

Step 4: Coefficient of Determination

$$R^2 = r^2 = (0.693)^2 \approx 0.48$$

Interpretation: The correlation coefficient $r \approx 0.693$ indicates a moderate positive relationship between x and y. The coefficient of determination $R^2 \approx 0.48$ suggests that about 48% of the variability in y can be explained by the linear relationship with x.

GitHub Repository: Introduction to ML

You can find all the code examples and datasets used in this report on GitHub at the following link:
`https://github.com/Naresh-812/Introduction_to_ML`

4.2.3 Gradient Descent Method

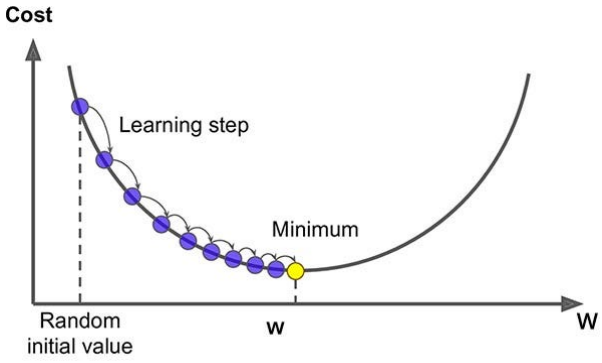


Figure 4.3: Gradient descent

From Statistics to Machine Learning

Gradient Descent is a powerful tool used in many machine learning algorithms, especially in linear regression and neural networks. But where does it come from?

A Statistical Origin: Minimizing Error

Let's say we have a simple linear model:

$$\hat{y} = b_0 + b_1 x$$

Our goal is to find the best values of b_0 and b_1 such that our predicted values $\hat{y}$ are as close as possible to the actual y values. One common way to measure this difference is using the Sum of Squared Errors (SSE):

$$J(b_0, b_1) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} (y_i - b_0 - b_1 x_i)^2$$

This cost function J comes directly from statistics. It tells us how "wrong" our predictions are. We want to find the parameters b_0 and b_1 that minimize this function.

How Do We Minimize It?

To minimize this function, we use derivatives to see which direction reduces the error the most — this is where gradient descent comes in.

Gradient Descent updates parameters by moving in the direction of the negative gradient (i.e., opposite to the slope):

$$b_0 := b_0 - \alpha \frac{\partial J}{\partial b_0}, \quad b_1 := b_1 - \alpha \frac{\partial J}{\partial b_1}$$

Where:

- α is the learning rate — it controls how big a step we take.
- $\frac{\partial J}{\partial b_0}$ and $\frac{\partial J}{\partial b_1}$ are the gradients (slopes) with respect to each parameter.

Computing the Gradients

To compute the gradients, we take partial derivatives of the cost function:

$$\frac{\partial J}{\partial b_0} = -2 \sum (y_i - b_0 - b_1 x_i)$$

$$\frac{\partial J}{\partial b_1} = -2 \sum x_i (y_i - b_0 - b_1 x_i)$$

Gradient Descent in Action

Think a Minute

Imagine standing on a mountain blindfolded (the cost function surface). The gradient tells you which direction is downhill. Taking small steps downhill repeatedly will eventually lead you to the bottom — the minimum error.

Summary

- Gradient Descent comes from trying to minimize the squared error — a statistical concept.
- It uses the idea of a gradient (slope) to update model parameters.
- It's widely used in training models where finding the analytical solution is hard or impossible.

Gradient Descent Calculation for Given Dataset

We are given:

$$x = [95, 85, 80, 70, 60], \quad y = [85, 95, 70, 65, 70]$$

We aim to fit a line:

$$\hat{y} = b_0 + b_1 x$$

Step 1: Initialize Parameters

- Learning rate: $\alpha = 0.0001$
- Initial values: $b_0 = 0$, $b_1 = 0$
- Number of epochs (iterations): 5 (for demonstration)

Step 2: Apply Gradient Descent Updates

Gradient Descent Step-by-Step Calculation (First 2 Iterations)

Given:

$$x = [95, 85, 80, 70, 60], \quad y = [85, 95, 70, 65, 70]$$

$$b_0 = 0, \quad b_1 = 0, \quad \alpha = 0.0001, \quad n = 5$$

The cost function to minimize:

$$J(b_0, b_1) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Gradients:

$$\frac{\partial J}{\partial b_0} = -\frac{2}{n} \sum (y_i - b_0 - b_1 x_i), \quad \frac{\partial J}{\partial b_1} = -\frac{2}{n} \sum x_i (y_i - b_0 - b_1 x_i)$$

$$y_i = actualvalue, \quad \hat{y}_i = predictedvalue$$

and used to update parameters:

$$b_1 = b_1 - \alpha \cdot \frac{\partial J}{\partial b_1}, \quad b_0 = b_0 - \alpha \cdot \frac{\partial J}{\partial b_0}$$

Iteration 1:

Predicted values:

$$\hat{y}_i = b_0 + b_1 x_i = 0 + 0 = 0 \text{ for all } i$$

Errors:

$$e_i = \hat{y}_i - y_i = -y_i \Rightarrow [-85, -95, -70, -65, -70]$$

Gradient for slope b_1:

$$\frac{2}{n}\sum(e_i \cdot x_i) = \frac{2}{5}[(-85)(95) + (-95)(85) + (-70)(80) + (-65)(70) + (-70)(60)]$$

$$= \frac{2}{5}[-8075 - 8075 - 5600 - 4550 - 4200] = \frac{2}{5}(-30500) = -12200$$

Gradient for intercept b_0:

$$\frac{2}{n}\sum e_i = \frac{2}{5}(-85 - 95 - 70 - 65 - 70) = \frac{2}{5}(-385) = -154$$

Update parameters:

$$b_1 = 0 - 0.0001 \cdot (-12200) = 1.22 \quad , \quad b_0 = 0 - 0.0001 \cdot (-154) = 0.0154$$

Iteration 2:

New predicted values:

$$\hat{y}_i = 0.0154 + 1.22 \cdot x_i = [116.1154,\ 104.7154,\ 98.6154,\ 85.4154,\ 72.2154]$$

Errors:

$$e_i = \hat{y}_i - y_i = [31.1154,\ 9.7154,\ 28.6154,\ 20.4154,\ 2.2154]$$

Slope gradient:

$$\frac{2}{5}\sum(e_i \cdot x_i) = \frac{2}{5}[(31.1154)(95) + (9.7154)(85) + (28.6154)(80) + (20.4154)(70) + (2.2154)(60)]$$

$$= \frac{2}{5}[2955.96 + 825.81 + 2289.23 + 1429.08 + 132.92] = \frac{2}{5}(7632.99) = 3053.20$$

Intercept gradient:

$$\frac{2}{5}\sum e_i = \frac{2}{5}(31.1154 + 9.7154 + 28.6154 + 20.4154 + 2.2154) = \frac{2}{5}(92.077) = 36.83$$

Update parameters:

$$b_1 = 1.22 - 0.0001 \cdot 3053.20 = 0.9147 \quad , \quad b_0 = 0.0154 - 0.0001 \cdot 36.83 = 0.0117$$

Note: Repeat similar steps for more iterations until convergence.

Updates over 5 Iterations

Epoch	b_0	b_1	$\frac{\partial J}{\partial b_0}$	$\frac{\partial J}{\partial b_1}$
1	0.0154	1.2200	-154.00	-12200.00
2	0.0117	0.9147	36.83	3053.20
3	0.0137	1.1223	-20.08	-2075.69
4	0.0124	0.9640	13.12	1583.40
5	0.0132	1.0807	-8.23	-1166.35

```
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999965
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999966
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999967
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999968
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999969
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999970
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999971
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999972
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999973
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999974
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999975
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999976
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999977
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999978
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999979
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999980
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999981
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999982
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999983
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999984
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999985
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999986
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999987
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999988
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999989
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999990
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999991
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999992
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999993
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999994
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999995
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999996
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999997
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999998
m 0.6438356164431029,b 26.780821917429087, cost 65.47945205479452,iterations 9999999
CPU times: user 4min 14s, sys: 5.29 s, total: 4min 20s
Wall time: 4min 32s
```

Figure 4.4: Gradent decent for 100000 iterations

$$m = slope, \quad b = intercept$$

Interpretation

With gradient descent, the algorithm gradually moves towards the optimal line that minimizes error. Even with a small learning rate and few iterations, we observe convergence toward the least square solution.

4.2.4 Stochastic Gradient Descent (SGD)

Definition

Stochastic Gradient Descent (SGD) is a variation of Gradient Descent used to minimize a cost (loss) function and optimize parameters such as weights in linear regression or neural networks.

How It Works

- **Batch Gradient Descent:** Uses the entire dataset to compute the gradient and update weights.
- **Stochastic Gradient Descent:** Updates weights using only **one sample at a time**, making it faster and suitable for large datasets.

Why "Stochastic"?

The term *stochastic* means "random." Since SGD uses random individual samples for each update, the learning path is noisy and irregular.

Advantages

- Faster updates for large datasets.
- Lower memory requirements.
- Can escape local minima.

Disadvantages

- Noisy updates (non-smooth convergence).
- May require more iterations.
- Sensitive to learning rate.

SGD vs Gradient Descent

Feature	Batch Gradient Descent	Stochastic Gradient Descent (SGD)
Data per step	All samples	One sample
Update speed	Slower	Faster
Convergence	Smooth	Noisy
Memory use	High	Low
Accuracy	Stable	Fluctuates

SGD Update Rule (Linear Regression)

Given one data point (x_i, y_i), the prediction is:

$$\hat{y}_i = b_0 + b_1 x_i$$

Error:

$$e_i = y_i - \hat{y}_i$$

Parameter updates:

$$b_1 = b_1 + \alpha \cdot e_i \cdot x_i$$

$$b_0 = b_0 + \alpha \cdot e_i$$

where α is the learning rate.

Python Library Example

Using `scikit-learn`:

```
from sklearn.linear_model import SGDRegressor

model = SGDRegressor(learning_rate='constant', eta0=0.01)
model.fit(X_train, y_train)
```

Gradient Descent Github Link

For a practical implementation and visualization of gradient descent from scratch and inbuilt, refer to the notebook available at:
`github.com/Naresh-812/Introduction_to_ML/gradent_descent.ipynb`

4.2.5 Multiple Linear Regression

Multiple Linear Regression (MLR) is an extension of simple linear regression that models the relationship between a dependent variable and multiple independent variables. The

general form of the model is:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p + \varepsilon$$

Where:

- y is the dependent variable,
- $x_1, x_2, \ldots, x_p$ are independent variables,
- β_0 is the intercept,
- $\beta_1, \beta_2, \ldots, \beta_p$ are the coefficients of the respective variables,
- ε is the error term.

The goal of MLR is to estimate the coefficients $\beta_0, \beta_1, \ldots, \beta_p$ such that the sum of squared residuals is minimized. This is typically done using the least squares method.

$$\min_{\beta} \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip}))^2$$

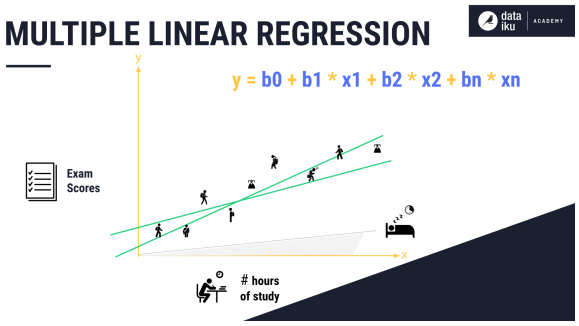


Figure 4.5: Multiple Linear regression

Applications

- Predicting house prices based on features like area, location, and number of rooms.
- Estimating student performance using study hours, sleep, and attendance.
- Forecasting sales using advertising budget, seasonality, and product rating.

Multicollinearity and Variance Inflation Factor (VIF)

What is Multicollinearity?

Multicollinearity occurs when two or more independent variables in a multiple linear regression model are highly correlated. This makes it difficult to determine the individual effect of each variable on the dependent variable.

Why is it a problem?

- It makes coefficient estimates unstable and highly sensitive to changes in the data.
- It reduces the interpretability of the model.
- It may inflate the standard errors of the coefficients.

Detecting Multicollinearity: Variance Inflation Factor (VIF)

The Variance Inflation Factor (VIF) is used to quantify the severity of multicollinearity.

VIF Formula:

$$\text{VIF}_i = \frac{1}{1 - R_i^2}$$

Where R_i^2 is the coefficient of determination obtained by regressing the i^{th} independent variable on all the other independent variables.

VIF Interpretation:

VIF Value	Interpretation
1	No multicollinearity
1 – 5	Low to moderate multicollinearity (acceptable)
5 – 10	High multicollinearity (needs attention)
> 10	Very high multicollinearity (consider removing variable)

What to Do If VIF is High?

- Remove or combine correlated variables.
- Use dimensionality reduction techniques such as PCA.
- Apply regularization methods like Ridge or Lasso regression.

Example

If you are predicting car price using `engine size`, `horsepower`, and `weight`, and `engine size` and `horsepower` are highly correlated, the VIF for these features may be high, indicating multicollinearity.

GitHub Notebook

You can view the full implementation of Multiple Linear Regression at the following GitHub link:
`github.com/Naresh-812/Introduction_to_ML/Multiple_linear_regression.ipynb`

4.3 Hyperparameters

What Are Hyperparameters?

Hyperparameters are settings that are chosen **before** training a machine learning model. They control the learning process, but **are not learned from the data**. Instead, they influence how the model learns from data.

Example: Think of hyperparameters like the settings on an oven (e.g., temperature, time) — they affect how the cake is baked but are not part of the ingredients.

Parameters vs. Hyperparameters

- **Parameters** (e.g., $\beta_0, \beta_1, \dots$) are learned from data.
- **Hyperparameters** (e.g., learning rate, number of iterations, regularization factor) are set manually before training.

Linear Regression

The standard linear regression equation is:

$$y = \beta_0 + \beta_1 x \quad \text{(Simple Linear Regression)}$$

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n \quad \text{(Multiple Linear Regression)}$$

In this case, $\beta_0, \beta_1, \dots, \beta_n$ are **parameters** learned by the model.

Hyperparameters in Linear Regression (When Using Gradient Descent)

- **Learning Rate** (α): Controls the step size in each update of gradient descent.
- **Number of Iterations**: Determines how many times the model updates its parameters.
- **Regularization Factor** (λ): Used in Ridge and Lasso regression to prevent overfitting.

Least Squares vs. Gradient Descent

Least Squares Method (Normal Equation):

$$\boldsymbol{\beta} = (X^T X)^{-1} X^T y$$

- Finds exact best-fit coefficients directly.
- Does not require hyperparameters.
- Fast for small datasets, but slow or infeasible for large ones.

Gradient Descent:

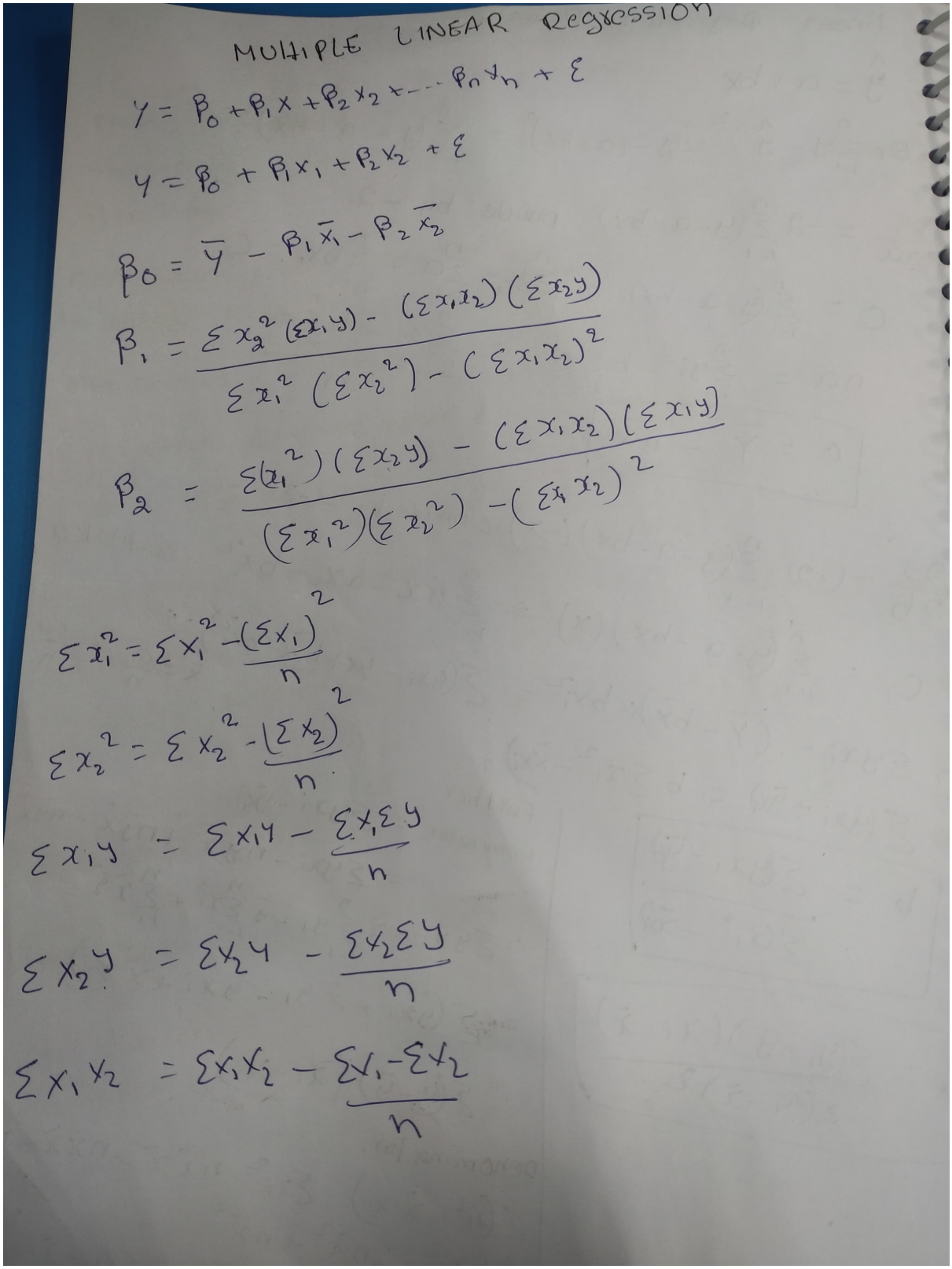
$$J(\beta) = \frac{1}{2m} \sum (h_\beta(x) - y)^2$$

- An iterative method to minimize the cost function.
- Requires hyperparameters like learning rate and number of iterations.

Summary Table

Concept	Parameters (Learned)	Hyperparameters (Set by User)
Simple Linear Regression	β_0, β_1	None (if using Least Squares)
Multiple Linear Regression	$\beta_0, \beta_1, \ldots, \beta_n$	None (least squares)
Gradient Descent	Coefficients updated iteratively	Learning rate α, Iterations
Regularized Linear Models	Coefficients β	Regularization factor λ

Mathematics behind Multiple Linear Regression (Two Variables)



MULTIPLE LINEAR Regression

$$y = \beta_0 + \beta_1 x + \beta_2 x_2 + \cdots \beta_n x_n + \varepsilon$$

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon$$

$$\beta_0 = \bar{y} - \beta_1 \bar{x}_1 - \beta_2 \bar{x}_2$$

$$\beta_1 = \frac{\Sigma x_2^2 (\Sigma x_1 y) - (\Sigma x_1 x_2)(\Sigma x_2 y)}{\Sigma x_1^2 (\Sigma x_2^2) - (\Sigma x_1 x_2)^2}$$

$$\beta_2 = \frac{\Sigma (x_1^2)(\Sigma x_2 y) - (\Sigma x_1 x_2)(\Sigma x_1 y)}{(\Sigma x_1^2)(\Sigma x_2^2) - (\Sigma x_1 x_2)^2}$$

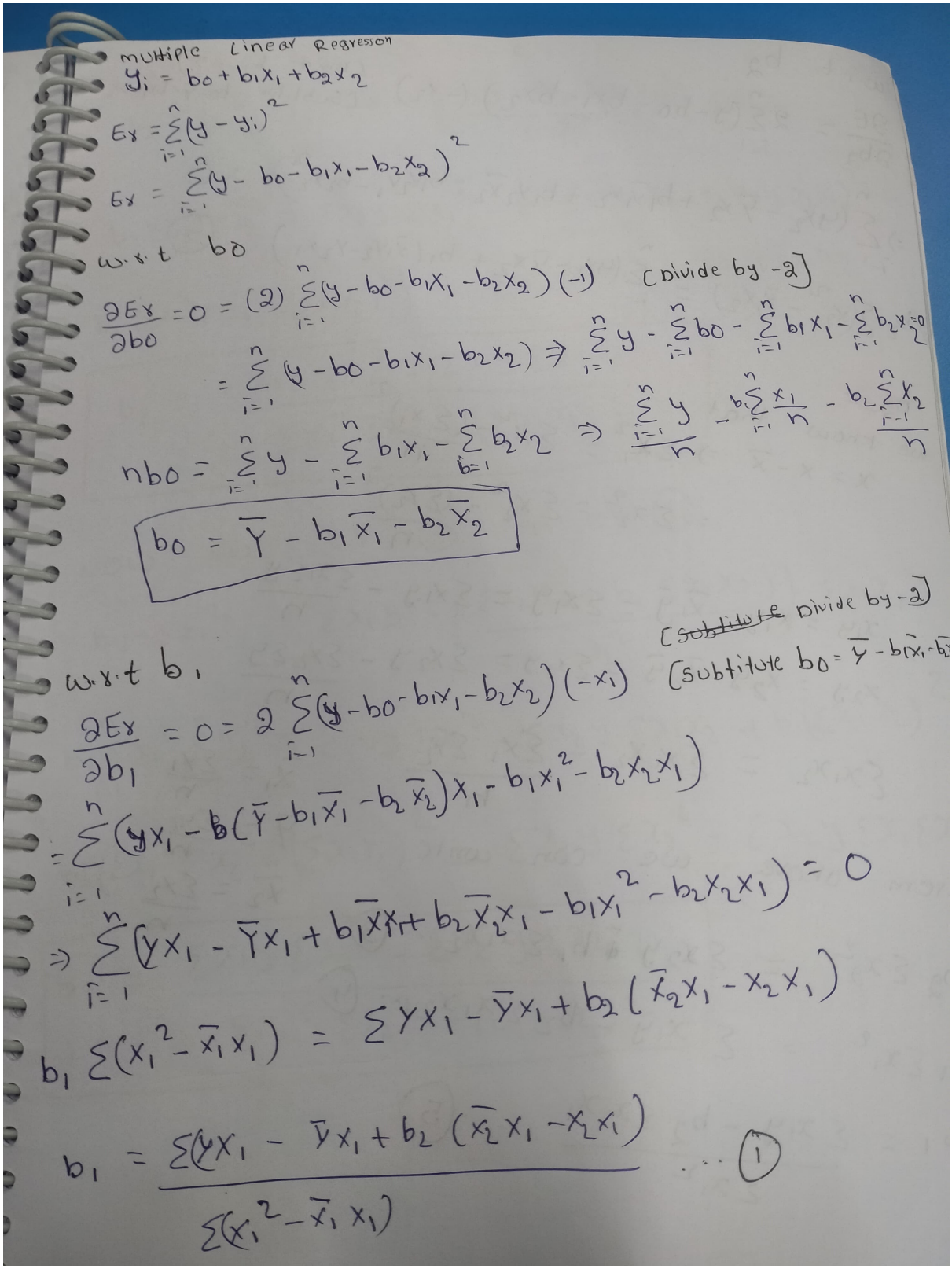
$$\Sigma x_1^2 = \Sigma x_1^2 - \frac{(\Sigma x_1)^2}{n}$$

$$\Sigma x_2^2 = \Sigma x_2^2 - \frac{(\Sigma x_2)^2}{n}$$

$$\Sigma x_1 y = \Sigma x_1 y - \frac{\Sigma x_1 \Sigma y}{n}$$

$$\Sigma x_2 y = \Sigma x_2 y - \frac{\Sigma x_2 \Sigma y}{n}$$

$$\Sigma x_1 x_2 = \Sigma x_1 x_2 - \frac{\Sigma x_1 - \Sigma x_2}{n}$$



Multiple Linear Regression

$y_i = b_0 + b_1 x_1 + b_2 x_2$

$E_r = \sum_{i=1}^{n} (y - y_i)^2$

$E_r = \sum_{i=1}^{n} (y - b_0 - b_1 x_1 - b_2 x_2)^2$

w.r.t b_0

$\frac{\partial E_r}{\partial b_0} = 0 = (2) \sum_{i=1}^{n} (y - b_0 - b_1 x_1 - b_2 x_2)(-1)$ [Divide by -2]

$= \sum_{i=1}^{n} (y - b_0 - b_1 x_1 - b_2 x_2) \Rightarrow \sum_{i=1}^{n} y - \sum_{i=1}^{n} b_0 - \sum_{i=1}^{n} b_1 x_1 - \sum_{i=1}^{n} b_2 x_2 = 0$

$nb_0 = \sum_{i=1}^{n} y - \sum_{i=1}^{n} b_1 x_1 - \sum_{b=1}^{n} b_2 x_2 \Rightarrow \frac{\sum_{i=1}^{n} y}{n} - b_1 \frac{\sum_{i=1}^{n} x_1}{n} - b_2 \frac{\sum_{i=1}^{n} x_2}{n}$

$\boxed{b_0 = \bar{Y} - b_1 \bar{x}_1 - b_2 \bar{x}_2}$

w.r.t b_1 [Subtitute ~~Subtitute~~ Divide by -2] [Subtitute $b_0 = \bar{Y} - b_1 \bar{x}_1 - b_2$

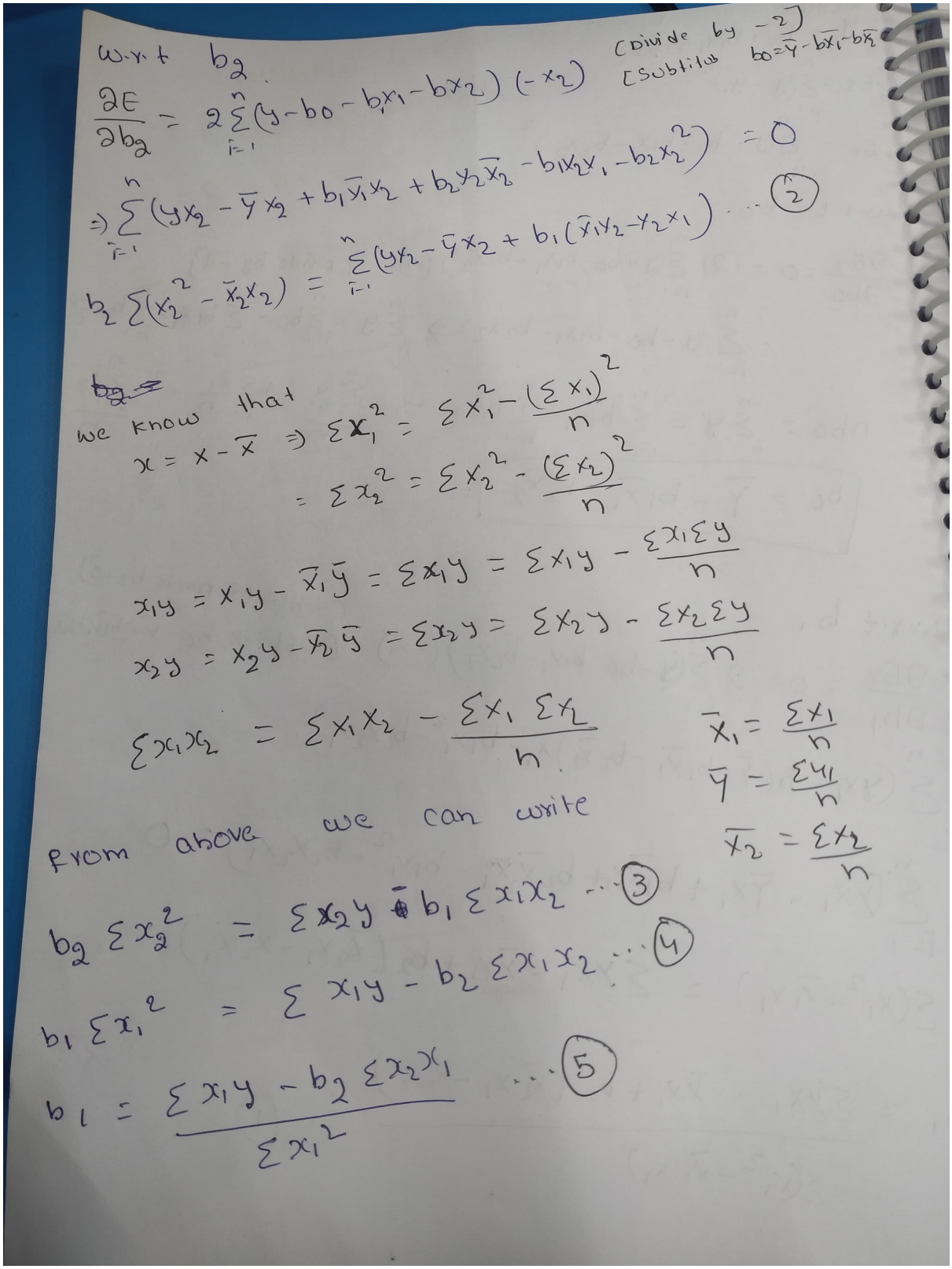
$\frac{\partial E_r}{\partial b_1} = 0 = 2 \sum_{i=1}^{n} (y - b_0 - b_1 x_1 - b_2 x_2)(-x_1)$

$= \sum_{i=1}^{n} (y x_1 - b_0 (\bar{Y} - b_1 \bar{x}_1 - b_2 \bar{x}_2) x_1 - b_1 x_1^2 - b_2 x_2 x_1)$

$\Rightarrow \sum_{i=1}^{n} (y x_1 - \bar{Y} x_1 + b_1 \bar{x}_1 x_1 + b_2 \bar{x}_2 x_1 - b_1 x_1^2 - b_2 x_2 x_1) = 0$

$b_1 \sum (x_1^2 - \bar{x}_1 x_1) = \sum y x_1 - \bar{y} x_1 + b_2 (\bar{x}_2 x_1 - x_2 x_1)$

$b_1 = \frac{\sum (y x_1 - \bar{y} x_1 + b_2 (\bar{x}_2 x_1 - x_2 x_1))}{\sum (x_1^2 - \bar{x}_1 x_1)} \quad \ldots (1)$



w.r.t b_2

$$\frac{\partial E}{\partial b_2} = 2\sum_{i=1}^{n}(y - b_0 - b_1x_1 - b_2x_2)(-x_2)$$ (Divide by -2) (Substitute $b_0 = \bar{y} - b\bar{x}_1 - b\bar{x}_2$)

$$\Rightarrow \sum_{i=1}^{n}(yx_2 - \bar{y}x_2 + b_1\bar{x}_1x_2 + b_2x_2\bar{x}_2 - b_1x_2x_1 - b_2x_2^2) = 0$$

$$b_2\sum(x_2^2 - \bar{x}_2x_2) = \sum_{i=1}^{n}(yx_2 - \bar{y}x_2 + b_1(\bar{x}_1x_2 - x_2x_1)) \quad \cdots (2)$$

we know that

$$x = x - \bar{x} \Rightarrow \Sigma x_1^2 = \Sigma x_1^2 - \frac{(\Sigma x_1)^2}{n}$$

$$= \Sigma x_2^2 = \Sigma x_2^2 - \frac{(\Sigma x_2)^2}{n}$$

$$x_1y = x_1y - \bar{x}_1\bar{y} = \Sigma x_1y = \Sigma x_1y - \frac{\Sigma x_1 \Sigma y}{n}$$

$$x_2y = x_2y - \bar{x}_2\bar{y} = \Sigma x_2y = \Sigma x_2y - \frac{\Sigma x_2 \Sigma y}{n}$$

$$\Sigma x_1x_2 = \Sigma x_1x_2 - \frac{\Sigma x_1 \Sigma x_2}{n}$$

$$\bar{x}_1 = \frac{\Sigma x_1}{n}$$

$$\bar{y} = \frac{\Sigma y_1}{n}$$

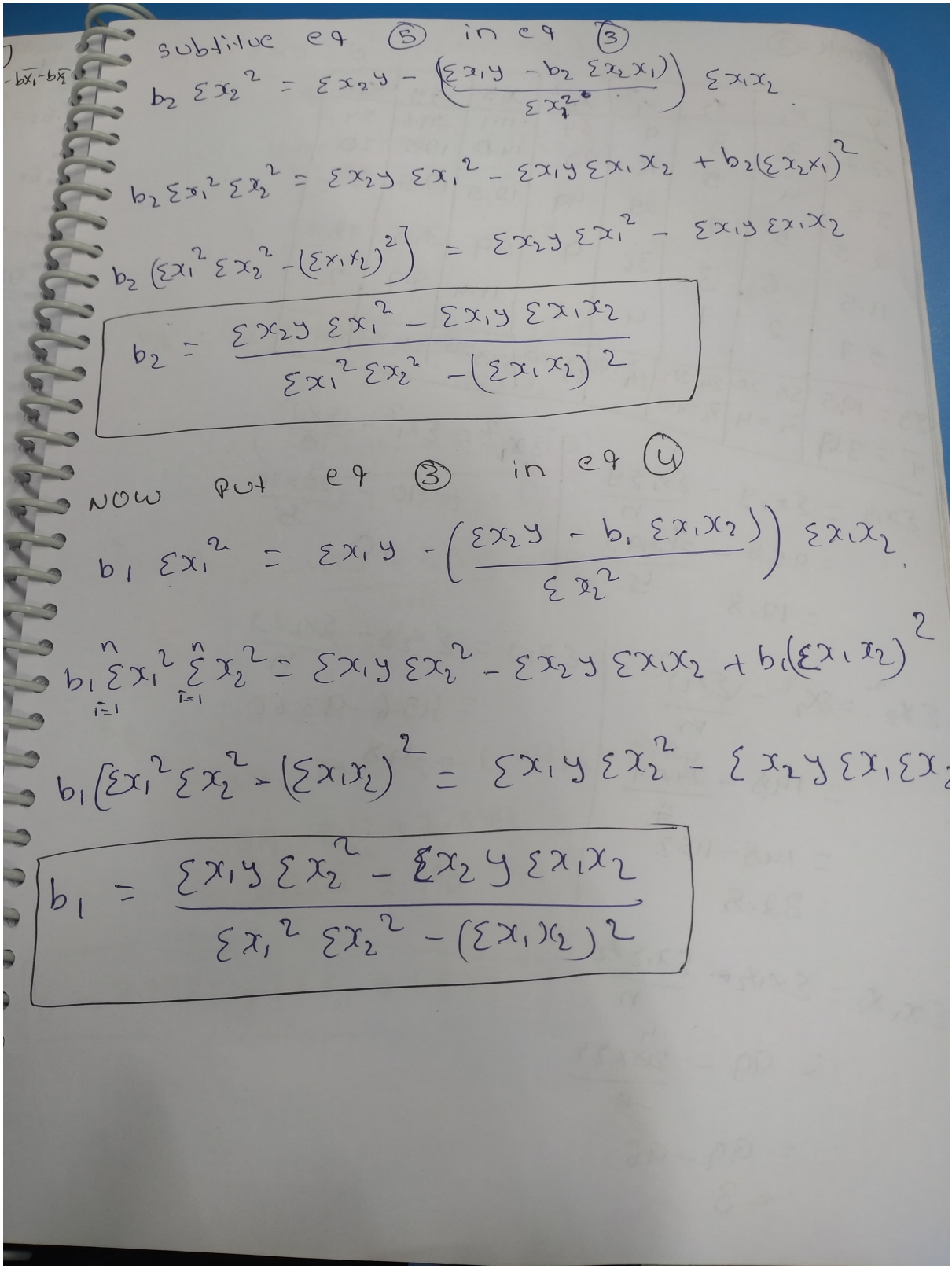
$$\bar{x}_2 = \frac{\Sigma x_2}{n}$$

From above we can write

$$b_2\Sigma x_2^2 = \Sigma x_2y - b_1\Sigma x_1x_2 \quad \cdots (3)$$

$$b_1\Sigma x_1^2 = \Sigma x_1y - b_2\Sigma x_1x_2 \quad \cdots (4)$$

$$b_1 = \frac{\Sigma x_1y - b_2\Sigma x_2x_1}{\Sigma x_1^2} \quad \cdots (5)$$



substitute eq (5) in eq (3)

$$b_2 \Sigma x_2^2 = \Sigma x_2 y - \left(\frac{\Sigma x_1 y - b_2 \Sigma x_2 x_1}{\Sigma x_1^2} \right) \Sigma x_1 x_2$$

$$b_2 \Sigma x_1^2 \Sigma x_2^2 = \Sigma x_2 y \Sigma x_1^2 - \Sigma x_1 y \Sigma x_1 x_2 + b_2 (\Sigma x_2 x_1)^2$$

$$b_2 \left(\Sigma x_1^2 \Sigma x_2^2 - (\Sigma x_1 x_2)^2 \right) = \Sigma x_2 y \Sigma x_1^2 - \Sigma x_1 y \Sigma x_1 x_2$$

$$b_2 = \frac{\Sigma x_2 y \Sigma x_1^2 - \Sigma x_1 y \Sigma x_1 x_2}{\Sigma x_1^2 \Sigma x_2^2 - (\Sigma x_1 x_2)^2}$$

Now put eq (3) in eq (4)

$$b_1 \Sigma x_1^2 = \Sigma x_1 y - \left(\frac{\Sigma x_2 y - b_1 \Sigma x_1 x_2}{\Sigma x_2^2} \right) \Sigma x_1 x_2$$

$$b_1 \sum_{i=1}^{n} x_1^2 \sum_{i=1}^{n} x_2^2 = \Sigma x_1 y \Sigma x_2^2 - \Sigma x_2 y \Sigma x_1 x_2 + b_1 (\Sigma x_1 x_2)^2$$

$$b_1 \left(\Sigma x_1^2 \Sigma x_2^2 - (\Sigma x_1 x_2)^2 \right) = \Sigma x_1 y \Sigma x_2^2 - \{ x_2 y \Sigma x_1 \Sigma x_2$$

$$b_1 = \frac{\Sigma x_1 y \Sigma x_2^2 - \Sigma x_2 y \Sigma x_1 x_2}{\Sigma x_1^2 \Sigma x_2^2 - (\Sigma x_1 x_2)^2}$$

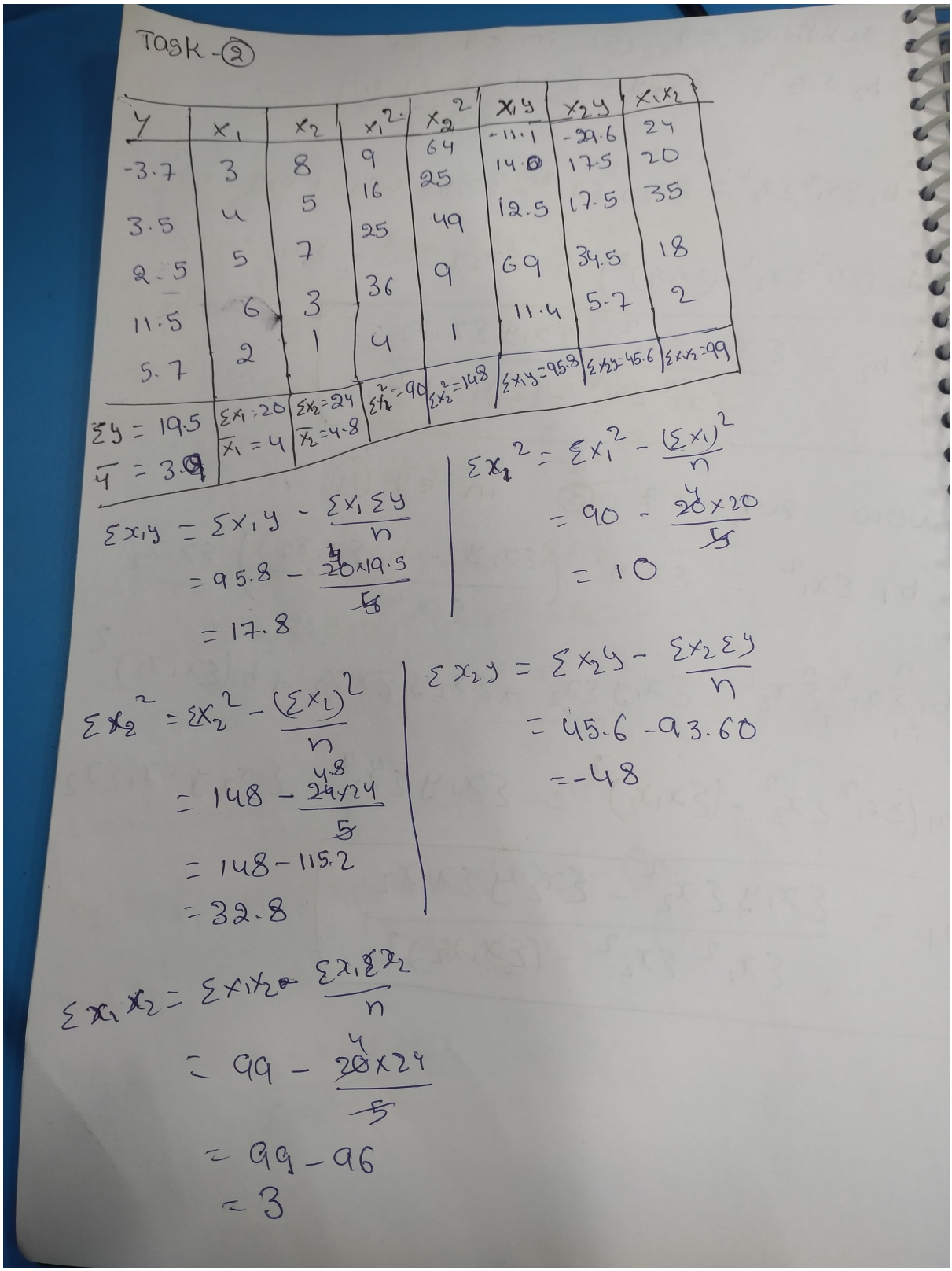
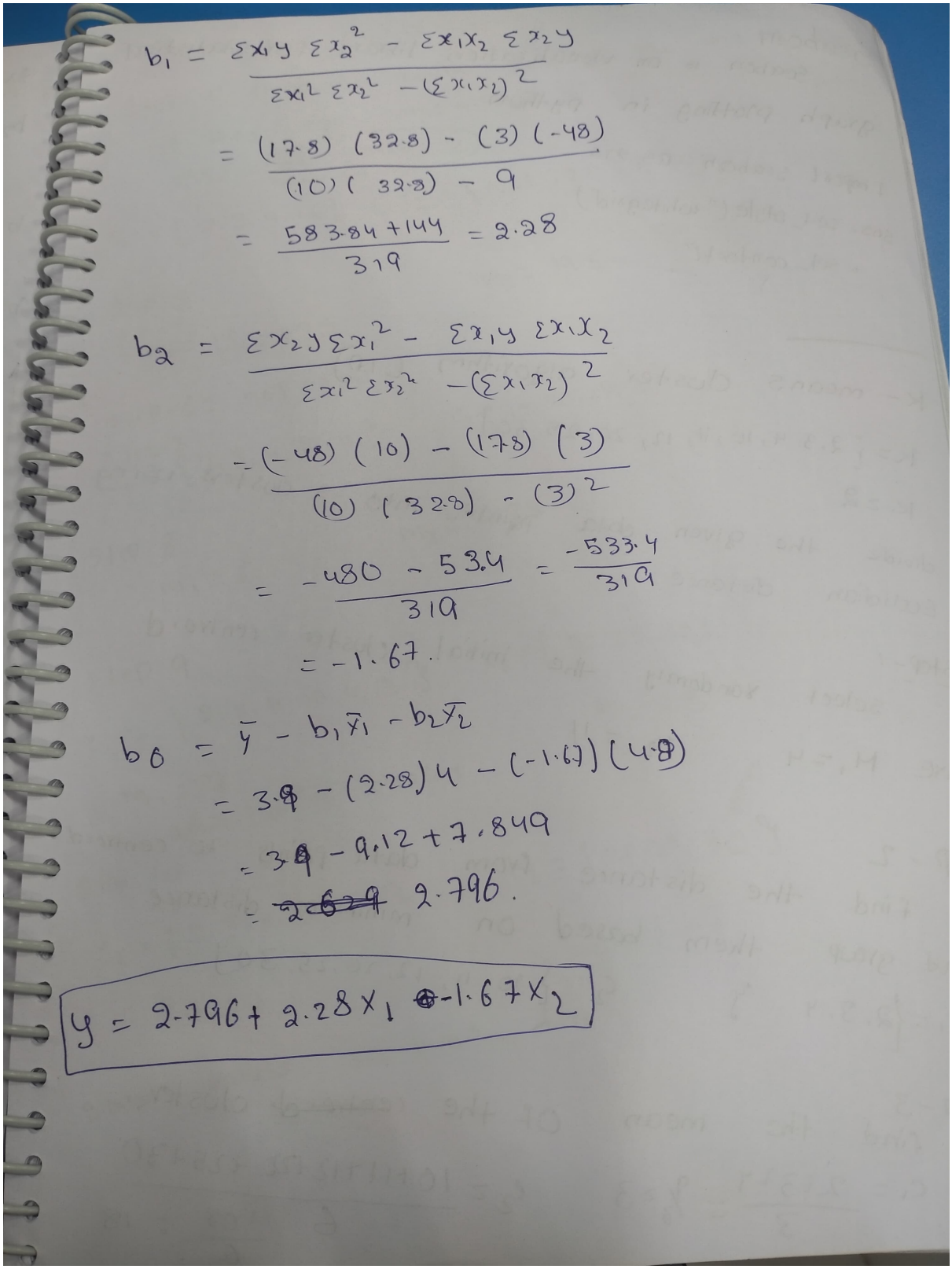


Figure 4.6: Example Problem



$$b_1 = \frac{\Sigma x_1 y \, \Sigma x_2^2 - \Sigma x_1 x_2 \, \Sigma x_2 y}{\Sigma x_1^2 \, \Sigma x_2^2 - (\Sigma x_1 x_2)^2}$$

$$= \frac{(17.8)(32.8) - (3)(-48)}{(10)(32.8) - 9}$$

$$= \frac{583.84 + 144}{319} = 2.28$$

$$b_2 = \frac{\Sigma x_2 y \, \Sigma x_1^2 - \Sigma x_1 y \, \Sigma x_1 x_2}{\Sigma x_1^2 \, \Sigma x_2^2 - (\Sigma x_1 x_2)^2}$$

$$= \frac{(-48)(10) - (17.8)(3)}{(10)(32.8) - (3)^2}$$

$$= \frac{-480 - 53.4}{319} = \frac{-533.4}{319}$$

$$= -1.67.$$

$$b_0 = \bar{y} - b_1 \bar{x}_1 - b_2 \bar{x}_2$$

$$= 3.8 - (2.28)4 - (-1.67)(4.8)$$

$$= 3.8 - 9.12 + 7.849$$

$$= 2.796.$$

$$y = 2.796 + 2.28 x_1 - 1.67 x_2$$

5. K-Means Clustering

Think a Minute

Suppose D-Mart plans to open a new store in a city. They want to pick a location that serves the maximum number of customers efficiently. How can they decide the best area to establish their store?

Which method would help them group neighborhoods or customers so they can make an informed decision?

This is where **K-Means Clustering** comes into play — a simple yet powerful way to group data points into meaningful clusters based on their similarity.

Intuitive Explanation: Imagine a city map dotted with the homes of customers. D-Mart wants to identify a few "central points" that represent groups of customers living near each other. By doing this, they can decide where to open stores so that each store serves customers in its cluster efficiently.

K-Means helps by dividing all these customer locations into K groups (clusters), where K is the number of stores or groups you want. Each group has a *centroid* — the average location of all customers in that cluster — which acts like the "center" of that group. K-Means tries to place these centroids so that the total distance between customers and their nearest centroid is as small as possible.

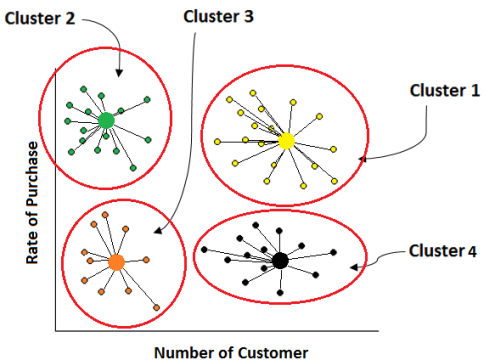


Figure 5.1: K-Means clustering

Technical Explanation: K-Means clustering is an iterative algorithm that partitions n data points $\{x_1, x_2, ..., x_n\}$ into K clusters $C = \{C_1, C_2, ..., C_K\}$ by minimizing the within-cluster sum of squares (WCSS), defined as:

$$J = \sum_{k=1}^{K} \sum_{x_i \in C_k} \|x_i - \mu_k\|^2$$

where:

- x_i is a data point,
- μ_k is the centroid (mean) of cluster C_k,
- $\|\cdot\|$ denotes Euclidean distance.

The algorithm follows these steps:

1. **Initialization:** Choose K initial centroids (can be random or via methods like K-Means++).
2. **Assignment step:** Assign each data point x_i to the nearest centroid μ_k.
3. **Update step:** Recalculate each centroid μ_k as the mean of all points assigned to cluster C_k.
4. Repeat steps 2 and 3 until convergence (when centroids no longer change significantly).

5.0.1 K-Means Clustering Example-1D

Given: Data points $\{2, 3, 4, 10, 11, 12, 20.25, 30\}$ and number of clusters $K = 2$.

Step 1: Initialize centroids Choose initial centroids m_1 and m_2. Let:

$$m_1 = 2, \quad m_2 = 30$$

Step 2: Assign each data point to the nearest centroid

Data Point	$\lvert x_i - m_1 \rvert$	$\lvert x_i - m_2 \rvert$	Assigned Cluster
2	$\lvert 2-2 \rvert = 0$	$\lvert 2-30 \rvert = 28$	1
3	$\lvert 3-2 \rvert = 1$	$\lvert 3-30 \rvert = 27$	1
4	$\lvert 4-2 \rvert = 2$	$\lvert 4-30 \rvert = 26$	1
10	$\lvert 10-2 \rvert = 8$	$\lvert 10-30 \rvert = 20$	1
11	$\lvert 11-2 \rvert = 9$	$\lvert 11-30 \rvert = 19$	1
12	$\lvert 12-2 \rvert = 10$	$\lvert 12-30 \rvert = 18$	1
20.25	$\lvert 20.25-2 \rvert = 18.25$	$\lvert 20.25-30 \rvert = 9.75$	2
30	$\lvert 30-2 \rvert = 28$	$\lvert 30-30 \rvert = 0$	2

Step 3: Calculate new centroids

$$m_1 = \frac{2+3+4+10+11+12}{6} = \frac{42}{6} = 7$$

$$m_2 = \frac{20.25+30}{2} = \frac{50.25}{2} = 25.125$$

Step 4: Reassign points based on updated centroids $m_1 = 7$ and $m_2 = 25.125$

Data Point	$\lvert x_i - m_1 \rvert$	$\lvert x_i - m_2 \rvert$	Assigned Cluster
2	$\lvert 2-7 \rvert = 5$	$\lvert 2-25.125 \rvert = 23.125$	1
3	$\lvert 3-7 \rvert = 4$	$\lvert 3-25.125 \rvert = 22.125$	1
4	$\lvert 4-7 \rvert = 3$	$\lvert 4-25.125 \rvert = 21.125$	1
10	$\lvert 10-7 \rvert = 3$	$\lvert 10-25.125 \rvert = 15.125$	1
11	$\lvert 11-7 \rvert = 4$	$\lvert 11-25.125 \rvert = 14.125$	1
12	$\lvert 12-7 \rvert = 5$	$\lvert 12-25.125 \rvert = 13.125$	1
20.25	$\lvert 20.25-7 \rvert = 13.25$	$\lvert 20.25-25.125 \rvert = 4.875$	2
30	$\lvert 30-7 \rvert = 23$	$\lvert 30-25.125 \rvert = 4.875$	2

Step 5: Calculate centroids again

$$m_1 = \frac{2+3+4+10+11+12}{6} = 7$$

$$m_2 = \frac{20.25+30}{2} = 25.125$$

Since cluster assignments and centroids did not change, the algorithm **converges** here.

Final clusters:

$$\begin{cases} \text{Cluster 1: } \{2,3,4,10,11,12\}, & m_1 = 7 \\ \text{Cluster 2: } \{20.25,30\}, & m_2 = 25.125 \end{cases}$$

5.0.2 Python code for Kmeans 1D

Without libraries

```
#k-means alogorith for 1d for 2 clusters
import numpy as np
data=np.array([2,3,4,10,11,12,20,25,30])
k=2#no of clusters
#initial centriodseeeeeeeeeeeeee
m1=data[0]
m2=data[-1]
c1_mean=0
c2_mean=0
while True:
#for calaculating distance and divinding clusters
  abs_dist1=np.abs(data-m1)
  abs_dist2=np.abs(data-m2)

  c1=[]
  c2=[]
  for i in range(len(data)):
    if abs_dist1[i]<abs_dist2[i]:#decideing which cluster it has to belongs
      c1.append(data[i])

    else:
      c2.append(data[i])

  c2=np.array(c2)

  c1=np.array(c1)
  c1_mean=c1.mean()
  c2_mean=c2.mean()
  if c1_mean==m1 and c2_mean==m2:
    print("done")
    break
  else:
    m1=c1_mean
    m2=c2_mean

print("cluster1=",c1)
print("cluster2=",c2)
print("m1=",m1)
print("m2=",m2)
```

Google Colab Notebook

You can access the interactive Google Colab notebook at:
https://colab.research.google.com/drive/17sVQ2DmNakrMJn-uDzEJZJwB-hj9w7_0?usp=sharing

K-Means Clustering for 2D Data

So far, we have seen K-Means clustering with one-dimensional data points. But in real life, data often comes in multiple dimensions. For example, imagine each data point represents the coordinates of a shop location on a map: (x, y).

In such cases, K-Means works similarly but uses the distance between points in 2D space (or higher dimensions) to group points into clusters.

How does K-Means work in 2D?

- **Initialize centroids:** Start by selecting K points in the 2D plane as initial centroids. Each centroid is represented as $\mathbf{m_i} = (x_i, y_i)$.
- **Assign points to clusters:** For each data point $\mathbf{p} = (x, y)$, calculate the **Euclidean distance** to each centroid:

$$d(\mathbf{p}, \mathbf{m_i}) = \sqrt{(x - x_i)^2 + (y - y_i)^2}$$

  Assign the point to the cluster whose centroid is closest.

- **Update centroids:** For each cluster, calculate the new centroid by averaging the x and y values of all points assigned to that cluster:

$$m_i = \left(\frac{1}{N_i} \sum_{j=1}^{N_i} x_j, \quad \frac{1}{N_i} \sum_{j=1}^{N_i} y_j \right)$$

  where N_i is the number of points in cluster i.

- **Repeat** the assignment and update steps until centroids no longer change significantly.

K-Means clustering in 2D (and higher dimensions) is widely used in image segmentation, customer segmentation, pattern recognition, and many other fields. It's a simple yet powerful way to discover natural groupings in complex data.

K-Means Clustering Example in 2D

Consider the following set of 2D data points:

$$(2, 3), (5, 6), (8, 7), (1, 4), (2, 2), (6, 7), (3, 4), (8, 6)$$

and the number of clusters $K = 3$.

Step 1: Initialize centroids

We select the first three points as initial centroids:

$$m_1 = (2, 3), \quad m_2 = (5, 6), \quad m_3 = (8, 7)$$

Step 2: Assign points to the nearest centroid

We calculate the Euclidean distance between each point $\mathbf{p} = (x, y)$ and each centroid $\mathbf{m} = (x_m, y_m)$:

$$d(\mathbf{p}, \mathbf{m}) = \sqrt{(x - x_m)^2 + (y - y_m)^2}$$

Point (x, y)	$d(\cdot, m_1)$	$d(\cdot, m_2)$	$d(\cdot, m_3)$	Assigned Cluster
(2,3)	0	$\sqrt{9+9} = 4.24$	$\sqrt{36+16} = 7.21$	1
(5,6)	4.24	0	$\sqrt{9+1} = 3.16$	2
(8,7)	7.21	3.16	0	3
(1,4)	$\sqrt{1+1} = 1.41$	$\sqrt{16+4} = 4.47$	$\sqrt{49+9} = 7.62$	1
(2,2)	$\sqrt{0+1} = 1$	$\sqrt{9+16} = 5$	$\sqrt{36+25} = 7.81$	1
(6,7)	$\sqrt{16+16} = 5.66$	$\sqrt{1+1} = 1.41$	$\sqrt{4+0} = 2$	2
(3,4)	$\sqrt{1+1} = 1.41$	$\sqrt{4+4} = 2.83$	$\sqrt{25+9} = 5.83$	1
(8,6)	$\sqrt{36+9} = 6.71$	$\sqrt{9+0} = 3$	$\sqrt{0+1} = 1$	3

Step 3: Update centroids

Calculate the new centroids by averaging the points assigned to each cluster:

$$m_1 = \left(\frac{2+1+2+3}{4}, \frac{3+4+2+4}{4} \right) = (2, 3.25)$$

$$m_2 = \left(\frac{5+6}{2}, \frac{6+7}{2} \right) = (5.5, 6.5)$$

$$m_3 = \left(\frac{8+8}{2}, \frac{7+6}{2} \right) = (8, 6.5)$$

Step 4: Re-assign points to nearest centroids with updated values

Point (x, y)	$d(\cdot, m_1)$	$d(\cdot, m_2)$	$d(\cdot, m_3)$	Assigned Cluster
(2,3)	$\sqrt{0 + 0.25} = 0.5$	$\sqrt{12.25 + 12.25} = 4.95$	$\sqrt{36 + 12.25} = 7.21$	1
(5,6)	$\sqrt{9 + 7.56} = 3.9$	$\sqrt{0.25 + 0.25} = 0.71$	$\sqrt{9 + 0.25} = 3.04$	2
(8,7)	$\sqrt{36 + 14.06} = 7.24$	$\sqrt{6.25 + 0.25} = 2.55$	$\sqrt{0 + 0.25} = 0.5$	3
(1,4)	$\sqrt{1 + 0.56} = 1.2$	$\sqrt{20.25 + 6.25} = 5.29$	$\sqrt{49 + 6.25} = 7.55$	1
(2,2)	$\sqrt{0 + 1.56} = 1.25$	$\sqrt{12.25 + 20.25} = 5.42$	$\sqrt{36 + 20.25} = 7.81$	1
(6,7)	$\sqrt{16 + 14.06} = 5.41$	$\sqrt{0.25 + 0.25} = 0.71$	$\sqrt{4 + 0.25} = 2.06$	2
(3,4)	$\sqrt{1 + 0.56} = 1.2$	$\sqrt{6.25 + 6.25} = 3.54$	$\sqrt{25 + 6.25} = 5.59$	1
(8,6)	$\sqrt{36 + 7.56} = 6.51$	$\sqrt{6.25 + 0.25} = 2.55$	$\sqrt{0 + 0.25} = 0.5$	3

Step 5: Calculate new centroids again

Cluster points have not changed, so centroids remain:

$$m_1 = (2, 3.25), \quad m_2 = (5.5, 6.5), \quad m_3 = (8, 6.5)$$

Since centroids did not change from the previous iteration, the algorithm converges.

Final clusters:

$$\begin{cases} \text{Cluster 1: } (2,3), (1,4), (2,2), (3,4) \\ \text{Cluster 2: } (5,6), (6,7) \\ \text{Cluster 3: } (8,7), (8,6) \end{cases}$$

Python Code for K-Means clustering without libraries (k=2)

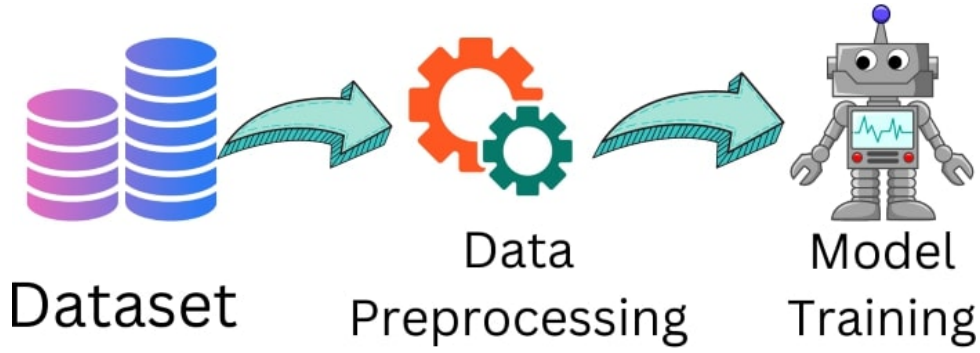
```
#k means for 2D arrays
import numpy as np
import matplotlib.pyplot as plt
data=np.array([[2,3],[5,6],[8,7],[1,4],[2,2],[6,7],[3,4],[8,6]])
m1=data[0]
m2=data[3]

while True:
  cl_1=[]
  cl_2=[]
  diff1=data-m1
  diff2=data-m2
  dist1=np.sqrt(np.sum(diff1**2,axis=1))
  dist2=np.sqrt(np.sum(diff2**2,axis=1))
  for i in range(len(data)):
    if dist1[i]<dist2[i]:
      cl_1.append(data[i])
    else:
      cl_2.append(data[i])
  cl_1=np.array(cl_1)
  print("one cluster=",cl_1)
  print("end")
  cl_2=np.array(cl_2)
  print("another cluster",cl_2)
  print("end")
  c1_mean=np.mean(cl_1,axis=0)
  print("c1 mean",c1_mean)
  c2_mean=np.mean(cl_2,axis=0)
  print("c2 mean",c2_mean)
  print("iteration completed")
  if np.array_equal(c1_mean, m1) and np.array_equal(c2_mean, m2):

    print("done")
    break
  else:
    m1=c1_mean
    m2=c2_mean
```

Google Colab Notebook

You can access the interactive Google Colab notebook at:
https://colab.research.google.com/drive/17sVQ2DmNakrMJn-uDzEJZJwB-hj9w7_0?usp=sharing

6. Data Preprocessing

6.1 Introduction to Data Preprocessing



Real-world data is often incomplete, inconsistent, or noisy, and cannot be directly used to train a machine learning model. Preprocessing involves cleaning and transforming the raw data into a form that a machine learning algorithm can understand and learn from.

Some common preprocessing tasks include:

- **Handling missing values:** Filling or removing data entries that are missing.
- **Normalization and scaling:** Ensuring all features are on the same scale.
- **Encoding categorical variables:** Converting text labels into numerical values.
- **Image preprocessing:** For image data, this may include resizing, normalization, or augmentation.

Think a Minute!

Imagine trying to train a model on pictures of cats and dogs where some images are tiny, some are huge, and others are blurry or labeled wrongly. Do you think the model can learn properly? That's why preprocessing is essential!

6.1.1 Handling Missing Values in Text Data

In real-world datasets, it's very common to encounter **missing values**. These are blank or 'NaN' entries in the dataset where some information is unavailable. For example, in a medical dataset, a patient might have skipped filling in their age or weight. Such gaps can confuse machine learning models and lead to poor predictions.

- **Removing rows or columns:** If too many values are missing, we can simply drop that row or column using functions like `dropna()` in Python.
- **Filling with mean/median/mode:** If numeric data is missing, we can fill it with the column's mean, median, or most frequent value (mode). For example:
  - Use `mean()` for normally distributed data.
  - Use `median()` for skewed data (e.g., income).
  - Use `mode()` for categorical features.
- **Filling with a constant value:** Sometimes, we can replace missing values with a constant like `0` or `"Unknown"`.
- **Forward/Backward fill:** In time-series or sequential data, we can fill missing values using the previous or next known value.

6.1.2 Example: Inserting and Handling Missing Values with Comments

Python Code

```
import pandas as pd
import numpy as np

# Step 1: Create a small dataset with missing values (NaN)
data = {
    'Name': ['Tom', 'Jerry', 'Spike', 'Tyke'],
    'Age': [20, np.nan, 25, np.nan],       # Age has missing values
    'Gender': ['M', 'M', np.nan, 'M']   # Gender has one missing value
}

# Convert to DataFrame
df = pd.DataFrame(data)
```

```
print("Original Data:")
print(df)

# Step 2: Fill missing values in the 'Age' column
# Calculates the mean of non-missing ages: (20 + 25) / 2 = 22.5
# Replaces all NaN in 'Age' with 22.5
df['Age'].fillna(df['Age'].mean(), inplace=True)

# Step 3: Fill missing values in the 'Gender' column
# Finds the most frequent value (mode), which is 'M'
# Replaces all NaN in 'Gender' with 'M'
df['Gender'].fillna(df['Gender'].mode()[0], inplace=True)

print("\nData After Filling Missing Values:")
print(df)
```

Output:

```
Original Data:
    Name   Age Gender
0    Tom  20.0      M
1  Jerry   NaN      M
2  Spike  25.0    NaN
3   Tyke   NaN      M

Data After Filling Missing Values:
    Name       Age Gender
0    Tom  20.000000      M
1  Jerry  22.500000      M
2  Spike  25.000000      M
3   Tyke  22.500000      M
```

This simple example shows how `fillna()` is used to handle missing data:

- **Numeric values (Age)** were replaced using the `mean()`.
- **Categorical values (Gender)** were filled using the `mode()`.

6.1.3 Encoding Categorical Variables

Machine learning algorithms work best with numbers. However, in real-world datasets, we often have **categorical data**, such as names of countries, product types, or yes/no values. These need to be converted into numerical format before training the model.

- **Label Encoding:** This assigns a unique number to each category.
  - Example: `[Red, Green, Blue]` → `[0, 1, 2]`
  - Simple but can confuse some models into thinking one label is greater than another.
- **One-Hot Encoding:** Creates a new column for each category and assigns 1 or 0 to indicate presence.
  - Example: `[Red, Green, Blue]` →

Red	Green	Blue
1	0	0
0	1	0
0	0	1

  - Prevents the algorithm from assuming an order between categories.
- **Ordinal Encoding:** Used when categories have a natural order (e.g., `[Low, Medium, High]` → `[1, 2, 3]`).

Think a Minute!

Suppose you have a dataset of animals with a "Type" column: `[Cat, Dog, Rabbit]`. If you label encode them as `[0, 1, 2]`, your model might think "Rabbit" is greater than "Cat". Would that make sense? That's why one-hot encoding is safer for unordered categories.

Choosing the right encoding technique depends on the nature of your data and the algorithm you are using. Some models handle categorical values better when properly encoded.

6.1.4 Example: One-Hot Encoding for Categorical Data

```
import pandas as pd
# Create a small dataset with a categorical column
data = {
    'Name': ['Tom', 'Jerry', 'Spike', 'Tyke'],
    'Pet_Type': ['Cat', 'Mouse', 'Dog', 'Dog']   # Categorical feature
}
```

```
df = pd.DataFrame(data)

print("Original Data:")
print(df)

# Apply One-Hot Encoding to 'Pet_Type' column
# This will create separate columns for each category (Cat, Dog, Mouse)
df_encoded = pd.get_dummies(df, columns=['Pet_Type'])

print("\nData After One-Hot Encoding:")
print(df_encoded)
```

Output:

```
Original Data:
    Name Pet_Type
0    Tom      Cat
1  Jerry    Mouse
2  Spike      Dog
3   Tyke      Dog

Data After One-Hot Encoding:
    Name  Pet_Type_Cat  Pet_Type_Dog  Pet_Type_Mouse
0    Tom             1             0               0
1  Jerry             0             0               1
2  Spike             0             1               0
3   Tyke             0             1               0
```

6.2 Image Data Preprocessing

Why is Image Preprocessing Important?

- Raw images have different resolutions, color formats, and noise levels.
- Models expect input images to be of a fixed shape and range (e.g., 224x224 pixels).
- Preprocessing enhances important features and reduces irrelevant variation.

Common Preprocessing Steps for Image Data:

1. **Resizing:** Adjust all images to a fixed size like 224x224 pixels.
2. **Normalization:** Scale pixel values (e.g., from [0, 255] to [0, 1]).
3. **Grayscale Conversion:** Reduce image to a single channel if color is not needed.
4. **Augmentation:** Apply transformations like rotation, flipping, or cropping to increase training diversity.
5. **Noise Removal:** Apply filters to smooth out the image and reduce sensor noise.

Think a Minute!

Did you know? Neural networks don't "see" images like humans do. To them, an image is just a matrix of numbers (pixel values). Preprocessing ensures that this matrix is structured in a way the model can understand and learn from effectively!

6.2.1 Working with the `os` Library in Python

When performing data augmentation on image datasets, we often need to read images from directories, create new folders for augmented images, or rename and save files. The `os` library in Python makes it easy to work with file paths and directory structures.

Why `os` is Important in Data Augmentation:

Commonly Used `os` Functions:

- `os.getcwd()` — Returns the current working directory.
- `os.listdir(path)` — Lists all files and folders in the specified path.
- `os.path.join(a, b)` — Joins folder and file names into a valid path.
- `os.makedirs(path)` — Creates a new directory (including intermediate folders if needed).
- `os.path.exists(path)` — Checks if a file or directory exists.
- `os.remove(file)` — Deletes a file.

```
import os
```

```
# Set input and output folders
input_folder = "dataset/original"
output_folder = "dataset/augmented"

# Create output folder if it doesn't exist
if not os.path.exists(output_folder):
    os.makedirs(output_folder)

# List all image files in the input folder
for filename in os.listdir(input_folder):
    if filename.endswith(".jpg") or filename.endswith(".png"):
        file_path = os.path.join(input_folder, filename)
        print("Processing:", file_path)
        # Load, augment, and save using other libraries like OpenCV
```

6.2.2 What is Data Augmentation?

Data Augmentation is a technique used to artificially increase the size and diversity of a dataset by creating modified versions of existing data. In the context of image processing, it involves applying transformations to original images to create new, realistic variants.

Why is Data Augmentation Important?

- Prevents overfitting by exposing the model to new variations.
- Helps in improving model generalization on unseen data.
- Useful when the available dataset is small or imbalanced.

Common Image Augmentation Techniques:

1. **Rotation** — Rotates the image by a few degrees.
2. **Flipping** — Horizontally or vertically flips the image.
3. **Cropping** — Randomly crops parts of the image.
4. **Scaling/Zooming** — Zooms in or out of the image.
5. **Translation** — Shifts the image left, right, up, or down.
6. **Brightness/Contrast Adjustment** — Simulates lighting changes.
7. **Noise Addition** — Adds small noise to improve robustness.

How it Works: Each time an image is used during training, a random transformation is applied, making the model see a slightly different version of the same image. This leads to a more flexible and better-performing model.

Think a Minute!

Even though you start with only 100 original images, you can train your model as if you had 1000 or more by using data augmentation. It's like teaching a model to recognize a cat whether it's upside down, sideways, or in bright sunlight!

6.2.3 Basic Syntax for Image Data Augmentation in Python

Data augmentation can be easily performed using libraries such as `TensorFlow/Keras`, `PIL`, or `OpenCV`. One of the most beginner-friendly tools is the `ImageDataGenerator` class from Keras.

Example Using `ImageDataGenerator` (Keras):

Python Code: Basic Data Augmentation

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Create an instance of ImageDataGenerator with augmentation parameters
datagen = ImageDataGenerator(
    rotation_range=20,        # Random rotation (degrees)
    width_shift_range=0.2,    # Horizontal shift
    height_shift_range=0.2,   # Vertical shift
    shear_range=0.1,          # Shear transformation
    zoom_range=0.2,           # Zoom in/out
    horizontal_flip=True,     # Flip image horizontally
    fill_mode='nearest'       # Fill in missing pixels
)
```

6.2.4 Python Script for Class-wise Image Augmentation

This script ensures that each class in the training dataset has exactly 1500 images by:

- Downsampling if a class has more than 1500 images.
- Augmenting if it has fewer.

Full Code with Explanation

Step 1: Importing Required Libraries

```
import os
import numpy as np
from tensorflow.keras.preprocessing.image import ImageDataGenerator,
    img_to_array, load_img, save_img
from tqdm import tqdm
import random
import shutil
```

- `os`: Handles file and directory paths.
- `numpy`: For numerical operations on image arrays.
- `tensorflow.keras.preprocessing.image`: Functions for image loading, processing, augmentation.
- `tqdm`: Shows progress bar.
- `random`: For randomizing image selection.
- `shutil`: Imported but unused; can help in file copying/moving.

Step 2: Set Paths and Constants

```
dataset_path = '/content/drive/MyDrive/Rice_Leaf_Diease/train'
target_size = 224
TARGET_IMAGES = 1500
```

- `dataset_path`: Path to the training dataset, with one folder per class.
- `target_size`: All images are resized to 224x224 pixels.
- `TARGET_IMAGES`: Each class should finally contain 1500 images.

Step 3: Define the Image Augmentation Generator

```
augmenter = ImageDataGenerator(
    rotation_range=15, width_shift_range=0.1,height_shift_range=0.1,
    zoom_range=0.2,horizontal_flip=True,
    fill_mode='nearest')
```

This will apply random transformations like rotation, shifting, zooming, and flipping.

Step 4: Iterate Through Each Class Folder

```
for class_name in os.listdir(dataset_path):
    class_path = os.path.join(dataset_path, class_name)
    if not os.path.isdir(class_path):
        continue
```

- Loops over each folder inside `train/`.
- Skips non-folder files.

Step 5: Count and Filter Images

```
    image_files = [f for f in os.listdir(class_path)
        if f.lower().endswith(('.png', '.jpg', '.jpeg'))]
    current_count = len(image_files)
```

- Filters only image files.
- `current_count` holds number of valid images.

Step 6: Handle Oversized Classes (Downsampling)

```
    if current_count > TARGET_IMAGES:
        ...
```

- If a class has more than 1500 images, it randomly deletes the extra ones:

```
        random.shuffle(image_files)
        keep_files = image_files[:TARGET_IMAGES]
        delete_files = set(image_files) - set(keep_files)
        for file_name in delete_files:
            os.remove(os.path.join(class_path, file_name))
```

Step 7: Skip Class if Already Balanced

```
    elif current_count >= TARGET_IMAGES:
        continue
```

Step 8: Calculate How Many New Images Are Needed

```
needed = TARGET_IMAGES - current_count
```

Step 9: Start Augmentation

```
i = 0
pbar = tqdm(total=needed, desc=f"Augmenting '{class_name}'")
```

- `pbar`: progress bar showing augmentation progress.

Step 10: Main Augmentation Loop

```
while i < needed:
    for img_name in image_files:
```

- Loops over the available images and augments until we get the required number.

Step 11: Load and Augment Image

```
img = load_img(img_path, target_size=(target_size, target_size))
x = img_to_array(img)
x = np.expand_dims(x, axis=0)
aug_iter = augmenter.flow(x, batch_size=1)
aug_img = next(aug_iter)[0].astype(np.uint8)
```

- Image is loaded and converted to a batch of size 1.
- `next(aug_iter)` generates one augmented image.

Step 12: Save Augmented Image

```
save_name = f"aug_{i}_{img_name}"
save_path = os.path.join(class_path, save_name)
save_img(save_path, aug_img)
i += 1
pbar.update(1)
```

Step 13: Exit If Done

```
if i >= needed:
    break
```

Step 14: Handle Exceptions Gracefully

```
        except Exception as e:
            print(f"Error processing {img_name}: {e}")
            continue
```

Step 15: Close Progress Bar

```
    pbar.close()
```

Conclusion

This script ensures every class folder in the training dataset contains the same number of images, either by removing excess or generating new ones. Such uniformity avoids class imbalance, making the model generalize better.

6.2.5 Dataset Splitting Script: Line-by-Line Explanation

We used a Python script to split the rice leaf disease image dataset into **training (70%)**, **validation (15%)**, and **testing (15%)** sets. Below is the code along with a detailed explanation for each step:

Dataset Splitting Script with Explanation

```
import os
import shutil
import random
from tqdm import tqdm
```

Explanation: These are standard Python modules used for:

- `os`: working with file paths and directories
- `shutil`: copying files
- `random`: shuffling image files
- `tqdm`: showing progress bars during file operations

```
original_dataset_dir = '/content/drive/MyDrive/Rice Leaf Diseases Dataset'
output_base_dir = '/content/drive/MyDrive/Rice Leaf Diseases Dataset_split'
```

Explanation: Defines the source and destination paths for the original dataset and the split dataset folders.

```
train_ratio = 0.7
val_ratio = 0.15
test_ratio = 0.15
```

Explanation: Sets the data split ratios: 70% for training, 15% for validation, and 15% for testing.

```
for split in ['train', 'val', 'test']:
    for class_name in os.listdir(original_dataset_dir):
        src = os.path.join(original_dataset_dir, class_name)
        if os.path.isdir(src):
            dest = os.path.join(output_base_dir, split, class_name)
            os.makedirs(dest, exist_ok=True)
```

Explanation:

- Iterates over the three splits and all class folders.
- Creates target directories like `train/blight`, `val/blight`, etc.
- `exist_ok=True` ensures it won't raise an error if the directory already exists.

```
for class_name in os.listdir(original_dataset_dir):
    class_dir = os.path.join(original_dataset_dir, class_name)
    if not os.path.isdir(class_dir):
        continue
```

Explanation: Loops through class directories and skips any non-folder files (e.g., metadata).

```
    files = [f for f in os.listdir(class_dir) if f.lower().endswith(('.jpg', '.png'
    random.shuffle(files)
```

Explanation:

- Filters only image files based on their extensions.
- Shuffles the list to ensure random sampling of images into each split.

```
    total = len(files)
    train_end = int(train_ratio * total)
    val_end = train_end + int(val_ratio * total)
```

Explanation: Calculates the exact number of images to go into each split based on the total count.

```
    splits = {
        'train': files[:train_end],
        'val': files[train_end:val_end],
        'test': files[val_end:]
    }
```

Explanation: Uses Python list slicing to partition images into train, validation, and test subsets.

```
    for split_name, split_files in splits.items():
        for file in tqdm(split_files, desc=f"{split_name.upper()}-{class_name}"):
            src_path = os.path.join(class_dir, file)
            dst_path = os.path.join(output_base_dir,split_name,class_name, file)
            shutil.copy(src_path, dst_path)
```

Explanation:

- Copies files from the original dataset into their respective split folders.
- Uses `tqdm` to display a live progress bar.

```
print("\nDataset successfully split into train, val, and test sets!")
```

Explanation: Prints a confirmation message once the dataset splitting is completed.

6.2.6 Image Data Generator Setup: Line-by-Line Explanation

We used `ImageDataGenerator` from Keras to preprocess and load the image data efficiently during model training, validation, and testing. Below is the annotated Python code with line-by-line explanation:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

Explanation: Imports the `ImageDataGenerator` class, which allows real-time data augmentation and preprocessing of image datasets.

```
img_size = 224
batch_size = 32
```

Explanation:

- `img_size` is the target image height and width (224x224) suitable for CNN models like VGG or MobileNet.
- `batch_size` defines how many images are fed into the network at once.

```
train_dir = '/content/drive/MyDrive/Rice_Leaf_Diease/train'
val_dir = '/content/drive/MyDrive/Rice_Leaf_Diease/validation'
test_dir = '/content/drive/MyDrive/Rice_Leaf_Diease/test'
```

Explanation: Specifies the directory paths for the training, validation, and test image folders. These folders are structured with subdirectories for each class.

```
train_val_datagen = ImageDataGenerator(
    rescale=1./255,
    validation_split=0.1
)
```

Explanation:

- `rescale=1./255` scales pixel values from the original range [0, 255] to [0, 1], which improves neural network performance.
- `validation_split=0.1` reserves 10% of training data for validation.

```
test_datagen = ImageDataGenerator(rescale=1./255)
```

Explanation: A separate `ImageDataGenerator` for the test set, with only rescaling (no augmentation or splitting).

```
train_generator = train_val_datagen.flow_from_directory(
    train_dir,
    target_size=(img_size, img_size),
    batch_size=batch_size,
    class_mode='categorical',
    subset='training',
    shuffle=True
)
```

Explanation:

- Loads the training data from the folder structure.

- `target_size` resizes all images to 224x224.
- `class_mode='categorical'` is used for multi-class classification.
- `subset='training'` fetches the training split from the 90% defined earlier.
- `shuffle=True` ensures the data is randomly shuffled during training.

```
val_generator = train_val_datagen.flow_from_directory(
    train_dir,
    target_size=(img_size, img_size),
    batch_size=batch_size,
    class_mode='categorical',
    subset='validation',
    shuffle=False
)
```

Explanation:

- Loads the validation split (10% of training data).
- `subset='validation'` ensures only validation data is accessed.
- `shuffle=False` maintains consistent ordering, which is useful for evaluation.

```
test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(img_size, img_size),
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False
)
```

Explanation:

- Loads test data from the separate test directory.
- No split is required; data is loaded directly.
- Shuffling is disabled to ensure reproducible results during testing.

Google Colab Notebook Link

https://colab.research.google.com/drive/1dAQq8c9H-GjdZV2VAhQ6fUCFsq0vzny2?usp=sharing

7. LeNet&Alexnet

7.1 LeNet-5

7.1.1 Introduction

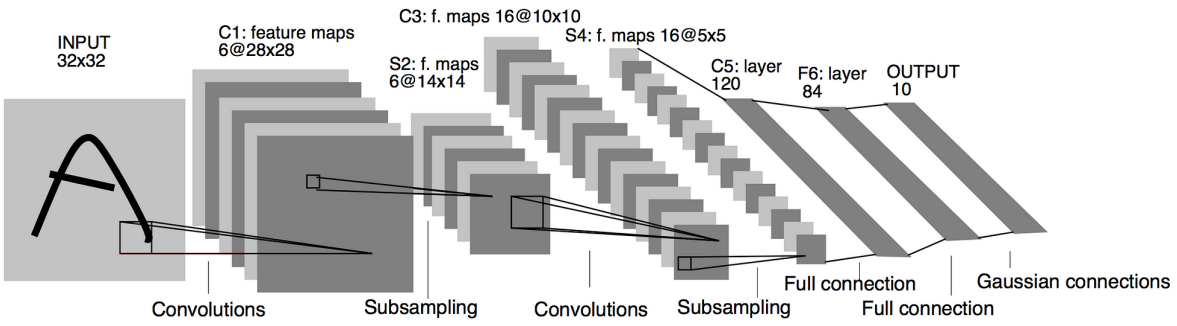
LeNet-5 is one of the earliest convolutional neural network (CNN) architectures, introduced by **Yann LeCun** in **1998** while working at Bell Labs. It was primarily developed for recognizing handwritten digits in bank checks and became foundational in the development of modern CNNs.

Motivation Behind LeNet

Before LeNet, image recognition relied on handcrafted features and traditional machine learning models. These were labor-intensive and performed poorly on large, diverse datasets. LeNet enabled automatic feature extraction using learnable convolutional filters, improving both accuracy and efficiency.

LeNet-5 Architecture

LeNet-5 is composed of seven layers, excluding the input. Each layer contains trainable parameters (weights and biases). The layers are as follows:



- **Input Layer**: Accepts 32×32 grayscale images.
- **C1 - Convolutional Layer**: Applies six 5×5 filters with stride 1.

- **S2 - Subsampling Layer (Average Pooling)**: Reduces each feature map using 2×2 average pooling with stride 2.
- **C3 - Convolutional Layer**: Applies sixteen 5×5 filters, each with specific connections to the previous layer.
- **S4 - Subsampling Layer**: Similar average pooling as S2.
- **C5 - Convolutional Layer**: Applies 120 filters of size 5×5.
- **F6 - Fully Connected Layer**: 84 neurons.
- **Output Layer**: 10 neurons (for digits 0-9), typically using softmax.

Activation Functions

LeNet-5 uses the **tanh** activation function. Modern CNNs prefer **ReLU** due to its computational efficiency and better gradient propagation:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \quad \text{ReLU}(x) = \max(0, x)$$

Mathematical Formulas and Calculations

Convolution Output Size: The output size after applying a convolution operation is calculated using the formula:

$$\text{Output Size} = \left(\frac{n + 2p - f}{s} \right) + 1$$

Where:

- n = input size
- p = padding
- f = filter size (kernel size)
- s = stride

Example: For the first convolutional layer (C1), with

$$n = 32, \quad p = 0, \quad f = 5, \quad s = 1$$

$$\text{Output Size} = \left(\frac{32 + 2 \times 0 - 5}{1} \right) + 1 = 28$$

So, the resulting output feature map is of size 28×28.

Pooling Output Size: The output size for a pooling (subsampling) layer is given by:

$$\text{Output Size} = \left(\frac{n-f}{s}\right) + 1$$

Where:

- n = input size
- f = pooling filter size
- s = stride

Example: For the first subsampling layer (S2), with

$$n = 28, \quad f = 2, \quad s = 2$$

$$\text{Output Size} = \left(\frac{28-2}{2}\right) + 1 = 14$$

So, the output feature map is 14×14 in size.

Parameter Calculation

- **C1:** $(5 \times 5 + 1) \times 6 = 156$ parameters
- **C3:** Total of 1516 parameters based on partial connections
- **C5:** $(5 \times 5 \times 16 + 1) \times 120 = 48,120$ parameters
- **F6:** $(120 + 1) \times 84 = 10,164$ parameters
- **Output Layer:** $(84 + 1) \times 10 = 850$ parameters

Total parameters: Approx. 60,806

Applications

- Handwritten digit recognition (MNIST dataset)
- Postal code and bank check processing
- Simple image classification tasks

Limitations

- Fixed input size (32×32)
- Uses tanh instead of modern activations like ReLU
- Lacks regularization (e.g., dropout, batch norm)
- Shallow depth limits learning of complex patterns

Comparison with Modern CNNs

Feature	**LeNet-5**	**Modern CNNs (e.g., ResNet, VGG)**
Depth	7 layers	50+ layers (deep networks)
Activation Function	`tanh`	`ReLU`, `Leaky ReLU`, `GELU`
Regularization	None	Dropout, Batch Normalization, Weight Decay
Input Flexibility	Fixed size 32×32	Flexible input sizes with adaptive pooling
Feature Learning	Shallow, limited feature extraction	Hierarchical, multi-level feature extraction
Performance	Suitable for simple tasks like digit recognition	State-of-the-art performance in vision tasks
Parameter Count	60K parameters	Millions of parameters
Hardware Use	Low resource requirement	Requires GPUs/TPUs for training

Table 7.1: Comparison between LeNet-5 and Modern Convolutional Neural Networks

Conclusion

LeNet-5 pioneered the use of convolutional layers for image recognition. Despite its simplicity, it laid the groundwork for modern deep learning models and remains an important architecture in the history of AI.

7.2 AlexNet

7.2.1 History and Motivation

AlexNet was introduced in 2012 by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. It significantly improved performance in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012, reducing the top-5 error rate from 26% to 15.3%. AlexNet proved the feasibility of training deep convolutional neural networks (CNNs) on large-scale datasets using GPU acceleration.

7.2.2 Architecture Overview

AlexNet's architecture contains 8 layers with learnable parameters:

- 5 Convolutional Layers (feature extraction)
- 3 Fully Connected Layers (classification)

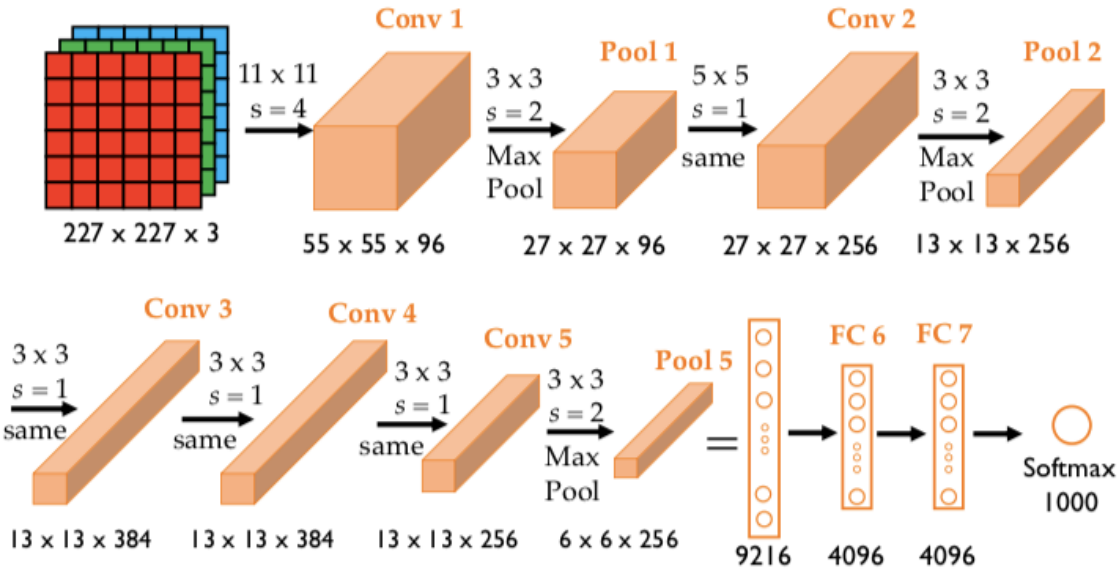


Figure 7.1: AlexNet Architecture

Layer-by-Layer Architecture

Key components:

- ReLU activation function
- Overlapping Max-Pooling

Layer	Type	Filters	Filter Size	Stride	Padding	Output Size
Input	-	-	-	-	-	$227 \times 227 \times 3$
Conv1	Conv + ReLU	96	11×11	4	0	$55 \times 55 \times 96$
Pool1	Max Pooling	-	3×3	2	-	$27 \times 27 \times 96$
LRN1	Norm	-	-	-	-	$27 \times 27 \times 96$
Conv2	Conv + ReLU	256	5×5	1	2	$27 \times 27 \times 256$
Pool2	Max Pooling	-	3×3	2	-	$13 \times 13 \times 256$
LRN2	Norm	-	-	-	-	$13 \times 13 \times 256$
Conv3	Conv + ReLU	384	3×3	1	1	$13 \times 13 \times 384$
Conv4	Conv + ReLU	384	3×3	1	1	$13 \times 13 \times 384$
Conv5	Conv + ReLU	256	3×3	1	1	$13 \times 13 \times 256$
Pool3	Max Pooling	-	3×3	2	-	$6 \times 6 \times 256$
FC6	Fully Connected	-	-	-	-	4096
FC7	Fully Connected	-	-	-	-	4096
FC8	Fully Connected + Softmax	-	-	-	-	1000

Table 7.2: AlexNet Architecture Layer-wise

- Local Response Normalization (LRN)
- Dropout for regularization
- Multi-GPU training

Mathematical Formulas

Convolution Output Size:

$$\text{Output Size} = \left(\frac{n + 2p - f}{s}\right) + 1$$

Where:

- n = input size
- p = padding
- f = filter size
- s = stride

Example: Conv1: $n = 227$, $f = 11$, $s = 4$, $p = 0$:

$$\left(\frac{227 - 11}{4}\right) + 1 = 55$$

Pooling Output Size:

$$\text{Output Size} = \left(\frac{n - f}{s}\right) + 1$$

Example: Pool1: $n = 55$, $f = 3$, $s = 2$:

$$\left(\frac{55-3}{2}\right) + 1 = 27$$

Key Innovations

- **ReLU Activation:** Faster convergence, avoids vanishing gradients.
- **Dropout:** Prevents overfitting by randomly omitting neurons.
- **Data Augmentation:** Improves generalization.
- **Local Response Normalization (LRN):** Enhances generalization.
- **GPU Parallelism:** Trains large models faster.

Comparison with LeNet-5

Feature	**LeNet-5**	**AlexNet**
Year	1998	2012
Input Size	32×32	227×227
Depth	7 layers	8 layers
Activation	Tanh	ReLU
Regularization	None	Dropout, LRN
GPU Support	No	Yes
Parameters	60K	60M
Application	Digit Recognition	Large-scale Image Classification

Table 7.3: Comparison of LeNet-5 and AlexNet

Applications

- Image Classification
- Object Detection (via R-CNN extensions)
- Medical Imaging Analysis
- Feature Extraction in Transfer Learning

Conclusion

AlexNet was a big breakthrough in the field of deep learning and computer vision. It showed that deep neural networks could solve difficult image classification problems better than traditional methods, especially when trained using powerful GPUs.

The use of new ideas like ReLU activation, dropout to prevent overfitting, and overlapping pooling helped improve the performance of the network. AlexNet's success inspired many researchers to build deeper and more advanced neural networks such as VGG, GoogLeNet, and ResNet.

In short, AlexNet played an important role in the growth of deep learning. It helped change how machines understand images and opened the door to many useful applications like medical image analysis, self-driving cars, and object detection.

7.2.3 coding part:

Objective

To implement and evaluate a deep learning model for **Mango Leaf Disease Classification** using the **AlexNet architecture**. The key objectives are to:

- Build the AlexNet model from scratch using `TensorFlow Keras`
- Train the model on a labeled mango leaf image dataset
- Classify images into multiple disease categories or healthy leaves
- Monitor model performance using accuracy, loss, precision, recall, and F1-score
- Evaluate the model using a confusion matrix and classification report
- Visualize training/validation loss and accuracy curves
- Improve performance through regularization and tuning hyperparameters

Implementation Workflow Using Python

Step 1: Import Required Libraries

```
1
2 !pip install split-folders
3
4 import numpy as np
5 import matplotlib.pyplot as plt
6 import os, zipfile
```

```
import random
import splitfolders
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import confusion_matrix, classification_report
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout,
    ↪ BatchNormalization
from tensorflow.keras.utils import to_categorical, plot_model
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import EarlyStopping
from IPython.display import Image, display
from keras.layers import GlobalAveragePooling2D
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint,
    ↪ ReduceLROnPlateau
```

Output:

```
Collecting split-folders
  Downloading split_folders-0.5.1-py3-none-any.whl.metadata (6.2 kB)
Downloading split_folders-0.5.1-py3-none-any.whl (8.4 kB)
Installing collected packages: split-folders
Successfully installed split-folders-0.5.1
```

Step 2: Load and Assign File Paths

```

from google.colab import drive
drive.mount('/content/drive')


zip_path = '/content/drive/MyDrive/mango_leaf_disease.zip'

dataset_path = 'mango_data_set'

with zipfile.ZipFile(zip_path, 'r') as zip_ref:
    zip_ref.extractall(dataset_path)

print("Extracted folders:", os.listdir(dataset_path))
```

Output:

```
Mounted at /content/drive
Extracted folders: ['Gall Midge', 'Die Back', 'Sooty Mould', 'Bacterial Canker',
'Healthy', 'Cutting Weevil', 'Anthracnose', 'Powdery Mildew']
```

Step 3: Split dataset into train/val/test

```

splitfolders.ratio('mango_data_set', output='splitted_mango_data_set', seed=42, ratio
    ↪ =(.7, .2, .1))

train_dir = 'splitted_mango_data_set/train'
val_dir = 'splitted_mango_data_set/val'
test_dir = 'splitted_mango_data_set/test'
```

Output:

```
Copying files: 4000 files [00:01, 3289.32 files/s]
```

Step 4: Print and Plot Class Distribution

```
def get_class_distribution(data_dir):
    return {
        cls: len(os.listdir(os.path.join(data_dir, cls)))
        for cls in os.listdir(data_dir)
        if os.path.isdir(os.path.join(data_dir, cls))
    }

train_dist = get_class_distribution(train_dir)
val_dist = get_class_distribution(val_dir)
test_dist = get_class_distribution(test_dir)

print("Training Data Counts:")
for cls, count in train_dist.items():
    print(f" - {cls}: {count} images")

print("valdation Data Counts:")
for cls, count in val_dist.items():
    print(f" - {cls}: {count} images")

print("Testing Data Counts:")
```

```
for cls, count in test_dist.items():
    print(f" - {cls}: {count} images")


plt.pie(train_dist.values(), labels=train_dist.keys(), autopct='%1.1f%%')
plt.title("Training Data Distribution")
plt.show()

plt.pie(val_dist.values(), labels=test_dist.keys(), autopct='%1.1f%%')
plt.title("validation Data Distribution")
plt.show()

plt.pie(test_dist.values(), labels=test_dist.keys(), autopct='%1.1f%%')
plt.title("Testing Data Distribution")
plt.show()
```

Output:

traingdatacount.png | valdatacount.png | testdatacount.png

Figure 7.2: Data distribution: (Left) Training, (Center) Validation, (Right) Testing

Step 5: Data Preprocessing and Augmentation

```

img_height, img_width = 227, 227
batch_size = 32
```

```

train_gen= ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
```

piediagtrain.png

```
    shear_range=0.2,
    zoom_range=0.3,
    horizontal_flip=True,
    fill_mode='nearest'
    ↪

)

val_gen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    zoom_range=0.3,
    horizontal_flip=True,
    shear_range=0.2,
    fill_mode='nearest')
test_gen = ImageDataGenerator(rescale=1./255)
```

Output:

```
Found 2800 images belonging to 8 classes.
```

```
Found 800 images belonging to 8 classes.
Found 400 images belonging to 8 classes.
```

Step 6: Load the Dataset

```

train_generator = train_gen.flow_from_directory(
    train_dir,
    target_size=(img_height, img_width),
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=True)

val_generator = val_gen.flow_from_directory(
    val_dir,
    target_size=(img_height, img_width),
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False)

test_generator = test_gen.flow_from_directory(
    test_dir,
    target_size=(img_height, img_width),
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False)
```

Step 7: Visualize Sample Input Data

```


class_labels = list(train_generator.class_indices.keys())

fig, axes = plt.subplots(ncols=4, nrows=4, figsize=(11,7))
for i in range(16):
    nrows=i//4
    ncols=i%4
    image, label = train_generator.__next__()
    axes[nrows,ncols].imshow(image[0])
    axes[nrows,ncols].set_title(f"{class_labels[np.argmax(label[0])]}")
    axes[nrows,ncols].axis('off')
plt.subplots_adjust(wspace=0.1, hspace=1)
plt.tight_layout()
plt.show()
```

```
\end{jupytercell}
\section*{Output:}
\begin{figure}[H]
    \centering
    \includegraphics[width=1\linewidth]{sampleinputs.png}
\end{figure}


\section*{Step 8: Define the Model}
\begin{jupytercell}{7}

model = Sequential()


model.add(Conv2D(filters=96, kernel_size=11, strides=4, activation='relu', input_shape
    ↪ =(227, 227, 3)))
model.add(MaxPooling2D(pool_size=3, strides=2))


model.add(Conv2D(filters=256, kernel_size=5, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=3, strides=2))


model.add(Conv2D(filters=384, kernel_size=3, padding='same', activation='relu'))


model.add(Conv2D(filters=384, kernel_size=3, padding='same', activation='relu'))


model.add(Conv2D(filters=256, kernel_size=3, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=3, strides=2))


model.add(Flatten())
model.add(Dense(units=4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(units=4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(units=train_generator.num_classes, activation='softmax'))


model.summary()
```

Output:

Step 9: Show Network Architecture

```
plot_model(model, to_file='alexnet_model_plot.png', show_shapes=True, show_layer_names
    ↪ =True, rankdir='LR')


try:
    display(Image(filename='alexnet_model_plot.png'))
except:
    print("Plot image saved as 'alexnet_model_plot.png'.")
```

Output:

Step 10: Compile the Model

```
from tensorflow.keras.optimizers import Adam
model.compile(loss='categorical_crossentropy', optimizer=Adam(learning_rate=0.0001),
    ↪ metrics=['accuracy'])

from keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau

early_stop = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
checkpoint = ModelCheckpoint("best_mango_model.h5", save_best_only=True, monitor="
    ↪ val_accuracy", mode="max")
lr_reduce = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=3, verbose=1)
```

Step 11: Train the Model

```
train = model.fit(train_generator, epochs=30, validation_data=val_generator,callbacks
    ↪ =[early_stop, lr_reduce, checkpoint])
```

Output:

Step 12: Model Prediction and Evaluation

```
1
2
3 test_images, test_labels = next(test_generator)
4 predictions = model.predict(test_images)
5 predicted_classes = np.argmax(predictions, axis=1)
6 true_classes = np.argmax(test_labels, axis=1)
7
8 score0 = model.evaluate(train_generator, verbose=0)
9 print('Training loss:', score0[0])
10 print('Training accuracy:', score0[1])
11 print("\n")
12 score1 = model.evaluate(val_generator, verbose=0)
13 print('validation loss:', score1[0])
14 print('validation accuracy:', score1[1])
15 print("\n")
16 score2 = model.evaluate(test_generator, verbose=0)
17 print('Testing loss:', score2[0])
18 print('Testing accuracy:', score2[1])
```

Output:

```
Training loss: 0.05553247034549713
Training accuracy: 0.9803571701049805

validation loss: 0.09699033945798874
validation accuracy: 0.96875

Testing loss: 0.04183173552155495
Testing accuracy: 0.9900000095367432
```

Step 13: Visualize Predicted Results

```
1
2
3 fig, axes = plt.subplots(ncols=5, figsize=(15, 4))
4 for i in range(5):
5     axes[i].imshow(test_images[i])
```

```
    axes[i].set_title(f"Pred: {class_labels[predicted_classes[i]]} \n actual: {
    ↪ class_labels[true_classes[i]]}")
    axes[i].axis('off')
plt.tight_layout()
plt.show()
```

Output:

Step 14: Plot Training History

```


plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(train.history['accuracy'], label="Train Accuracy")
plt.plot(train.history['val_accuracy'], label="Validation Accuracy")
plt.title("Model Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(train.history['loss'], label="Train Loss")
plt.plot(train.history['val_loss'], label="Validation Loss")
plt.title("Model Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.tight_layout()
plt.show()
```

Output:

Step 15: Confusion Matrix and Classification Report

```

from sklearn.metrics import classification_report, confusion_matrix, f1_score
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

y_pred_probs = model.predict(test_generator)
```

```
y_pred = np.argmax(y_pred_probs, axis=1)


y_true = test_generator.classes
class_labels = list(test_generator.class_indices.keys())

print("Classification Report:")
print(classification_report(y_true, y_pred, target_names=class_labels))

cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=class_labels, yticklabels=class_labels)
plt.title("Confusion Matrix")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()

f1 = f1_score(y_true, y_pred, average='macro')
print(f"Macro F1 Score: {f1:.4f}")
```

Output:

```
13/13  20s 1s/step
Classification Report:
                 precision    recall  f1-score   support

     Anthracnose       1.00      1.00      1.00        50
Bacterial Canker       1.00      1.00      1.00        50
  Cutting Weevil       1.00      1.00      1.00        50
        Die Back       1.00      1.00      1.00        50
      Gall Midge       1.00      1.00      1.00        50
         Healthy       1.00      1.00      1.00        50
  Powdery Mildew       0.93      1.00      0.96        50
    Sooty Mould        1.00      0.92      0.96        50

        accuracy                           0.99       400
       macro avg       0.99      0.99      0.99       400
    weighted avg       0.99      0.99      0.99       400
```

Step 16: Save the Model

```

temp_save_path = "Alexnet_mangoleaf_disease_classification(flatten_30_epochs).h5"
model.save(temp_save_path)


drive_save_path = "/content/drive/MyDrive/Nitw/
    ↪ Alexnet_mangoleaf_disease_classification(flatten_30_epochs).h5"


drive_save_dir = os.path.dirname(drive_save_path)


if not os.path.exists(drive_save_dir):
    os.makedirs(drive_save_dir)
    print(f"Created directory: {drive_save_dir}")


import shutil
shutil.copyfile(temp_save_path, drive_save_path)
print(f"Model saved successfully to {drive_save_path}")


import os
os.remove(temp_save_path)
```

Step 17: Load and Use Saved Model (Optional)

```


    ↪
loaded_model = load_model(drive_save_path)
score_loaded = loaded_model.evaluate(test_generator, verbose=0)
print('Loaded model testing loss:', score_loaded[0])
print('Loaded model testing accuracy:', score_loaded[1])
```

piediagval.png

piediagtest.png

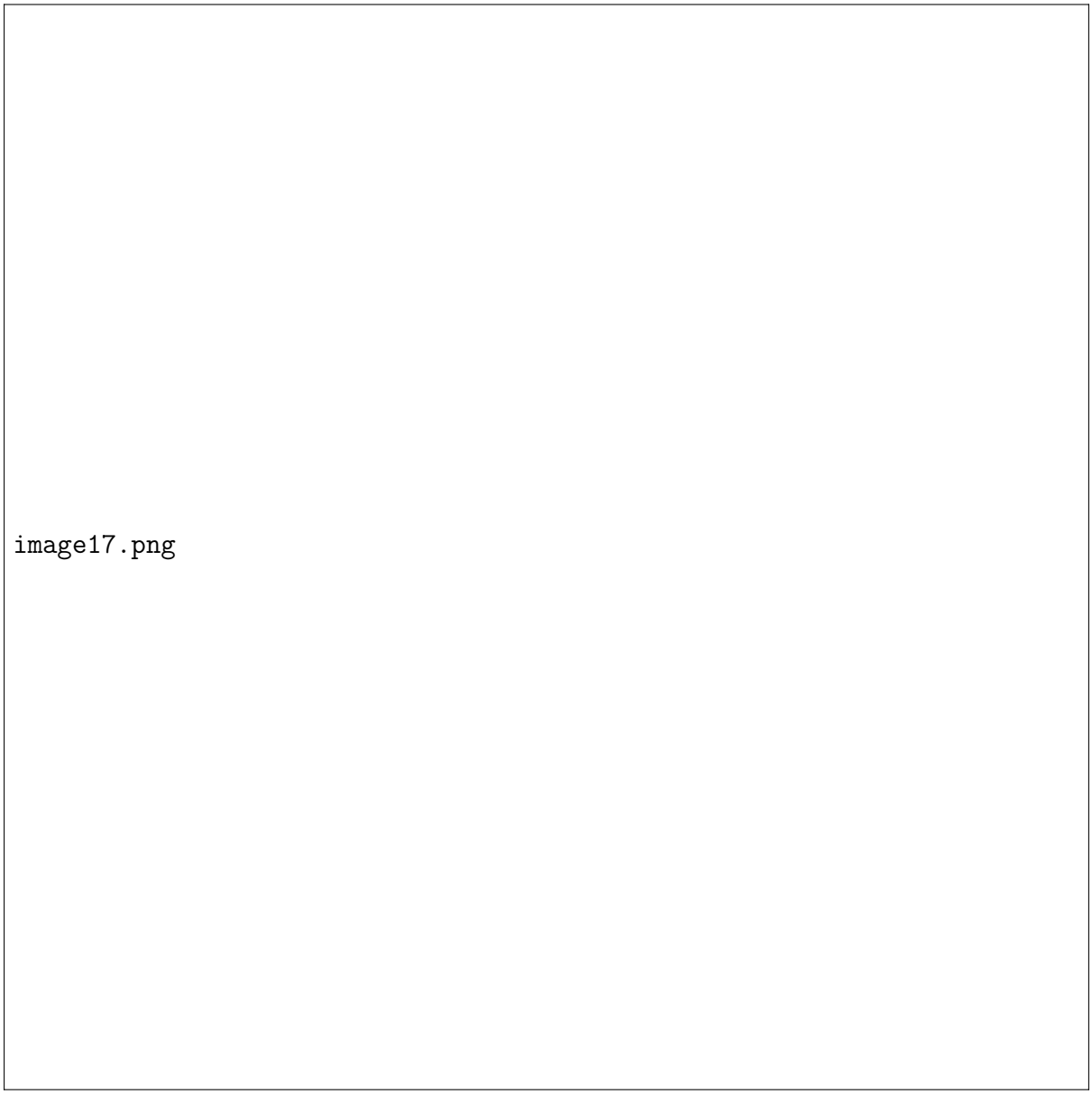
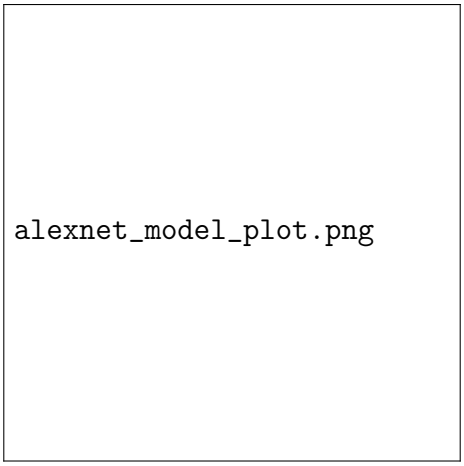


image17.png


alexnet_model_plot.png

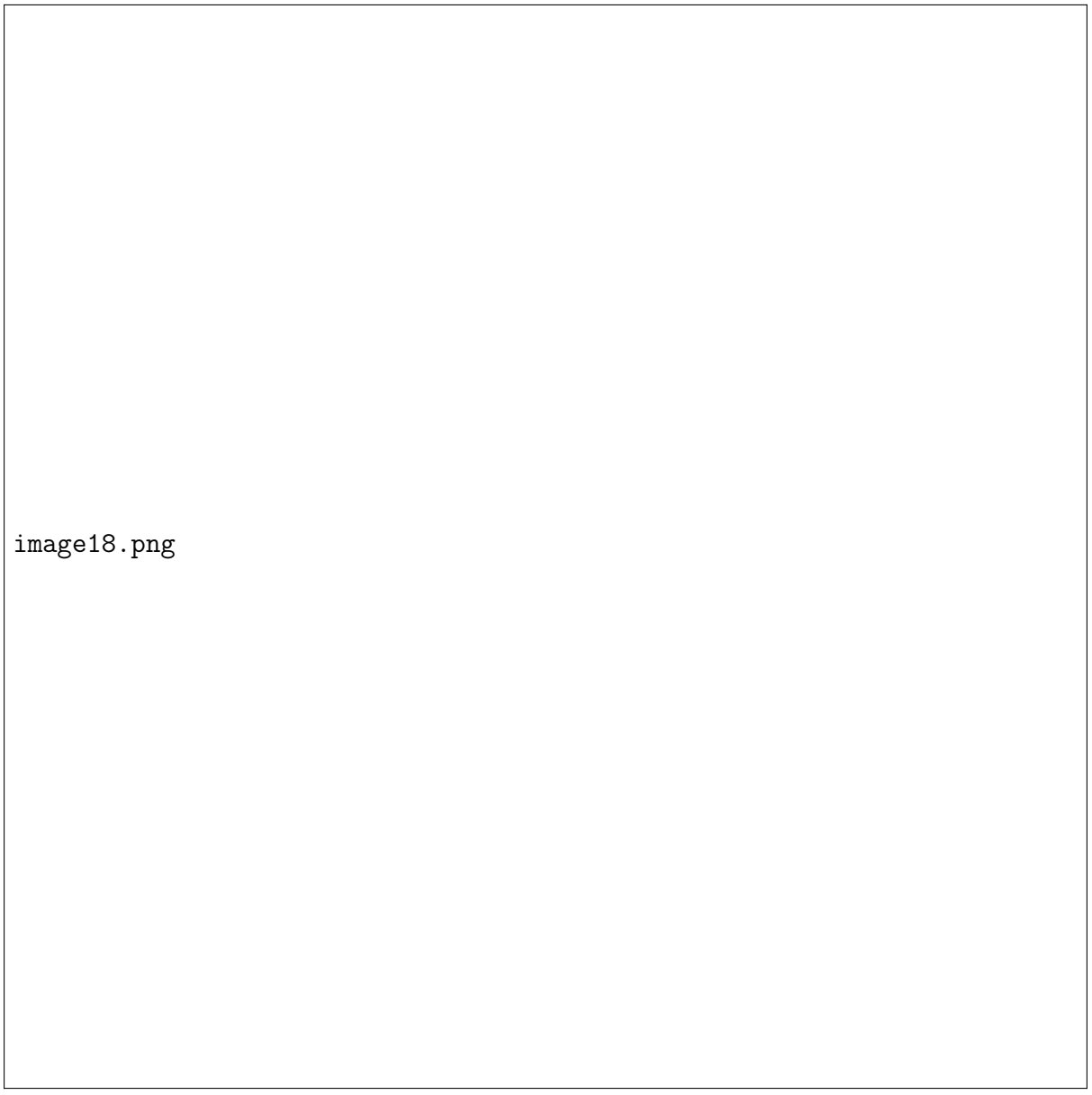


image18.png

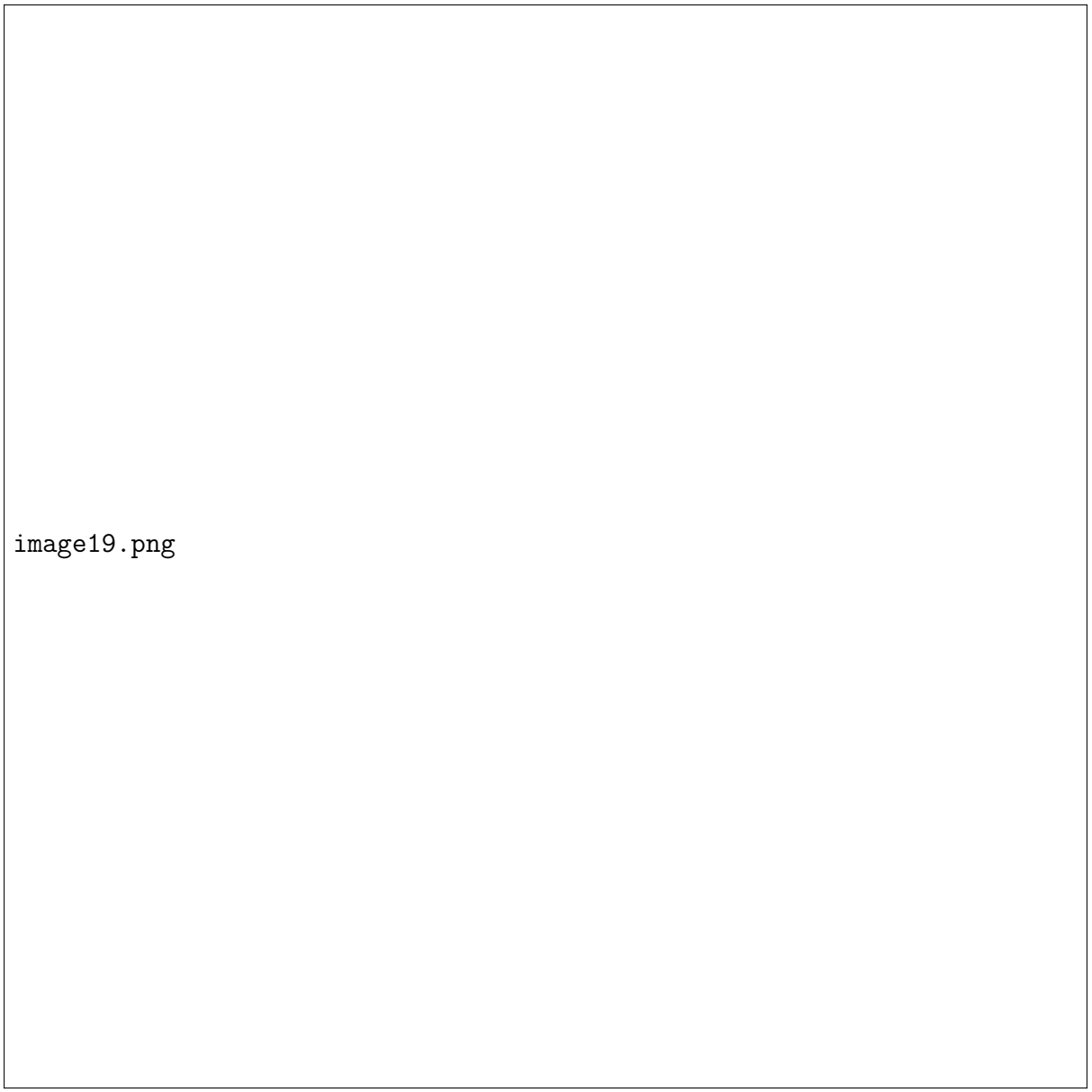


image19.png

image20.png

image21.png

8. VGG16-Architecture

8.1 Introduction

VGG-16 is a popular deep convolutional neural network (CNN) architecture that was proposed by the Visual Geometry Group (VGG) at the University of Oxford in 2014. It was introduced in the research paper titled *"Very Deep Convolutional Networks for Large-Scale Image Recognition"* by Karen Simonyan and Andrew Zisserman. VGG-16 is a type of CNN that is considered one of the most effective computer vision models to date. This model increased the network depth using an architecture with very small (3 $\times$ 3) convolution filters, which showed significant improvement.

VGG-16 is an object detection and classification algorithm that can classify images into 1000 different categories with 92.7% accuracy. It is one of the most popular algorithms for image classification and is easy to use with transfer learning.

We shifted from AlexNet to VGG-16 because VGG-16 offered a deeper, cleaner, and more powerful architecture that produced significantly better results on large-scale image classification tasks. VGG-16 significantly outperformed AlexNet on the ImageNet classification task, with higher top-1 and top-5 accuracy. The deeper structure of VGG-16 allows it to learn richer and more detailed features, which are especially useful for transfer learning.

VGG-16 uses a repeating pattern (Conv $\rightarrow$ ReLU $\rightarrow$ Conv $\rightarrow$ ReLU $\rightarrow$ MaxPool), which is easier to understand and modify than the more irregular structure of AlexNet.

Architecture

Coming to the architecture of VGG16 16 in VGG16 refers to 16 layers that have weights. In VGG16 there are thirteen convolutional layers, five Max Pooling layers, and three Dense layers which sum up to 21 layers but it has only sixteen weight layers i.e., learnable parameters layer.VGG16 takes input tensor size as 224, 244 with 3 RGB

channel.Most unique thing about VGG16 is that instead of having a large number of hyper-parameters they focused on having convolution layers of 3x3 filter with stride 1 and always used the same padding and maxpool layer of 2x2 filter of stride 2.

The convolution and max pool layers are consistently arranged throughout the whole architecture Conv-1 Layer has 64 number of filters, Conv-2 has 128 filters, Conv-3 has 256 filters, Conv 4 and Conv 5 has 512 filters.Three Fully-Connected (FC) layers follow a stack of convolutional layers: the first two have 4096 channels each, the third performs 1000-way ILSVRC classification and thus contains 1000 channels (one for each class). The final layer is the soft-max layer.

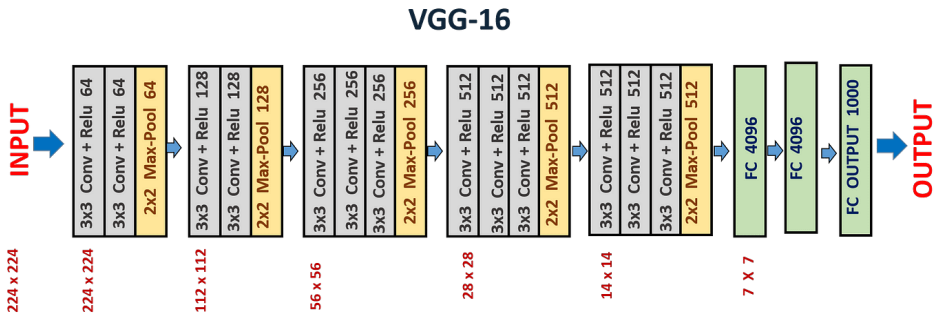


Figure 8.1: Architecture of VGG16

Sytax to import VGG16

```
from tensorflow.keras.applications import VGG16


model = VGG16(weights=None,
              input_shape=(224, 224, 3),
              include_top=True,
              classes=1000)
```

we can import VGG16 model directly from tensorflow.keras.applications library

8.1.1 Transfer Learning using VGG16

Transfer learning with VGG16 involves using its pre-trained weights from ImageNet to solve a new classification task. The early layers of VGG16 are kept (as feature extractors), while the top layers i.e *dense layers* are replaced with custom layers for the new dataset. This approach works well even with limited training data. It significantly reduces training time and improves accuracy on tasks like medical imaging, plant disease detection, or object recognition.

We are using this transfer learnig for better accuracy and also by unfreezing some layers we can also get more accuracy for our dataset

VGG 16 implementation using Transfer learning

Listing 8.1: Importing necessary libraries

```
1
2  from time import perf_counter
3  start = perf_counter()
4  import numpy as np
5  from tensorflow.keras.preprocessing.image import ImageDataGenerator
6  from tensorflow.keras.applications import VGG16
7  from tensorflow.keras.models import Model
8  from tensorflow.keras.layers import Flatten, Dense, Dropout, BatchNormalization
9  from tensorflow.keras.optimizers import Adam
10 from sklearn.metrics import classification_report
11 import matplotlib.pyplot as plt
12 from tensorflow.keras.utils import plot_model
13 from IPython.display import Image, display
```

From tensorflow.keras import all the necessary libraries that are required to implement VGG16 such as

- ImageData Generators for data preprocessing
- Model to implement our model, Optimizers such as adam,sgd
- All the necessary layers such as Flatten,Dense,Dropout,Batchnormalization
- classification report to check the metrices
- plot_model lets you visualize the architecture of a Keras model as a diagram.
- Image,display from IPython.display will display the image file vgg16_architecture.png inside the notebook.

Listing 8.2: Giving dataset paths and data preprocessing using ImageDataGenerator

```

train_dir = '/content/drive/MyDrive/Colab Notebooks/grape_dataset/train'
test_dir = '/content/drive/MyDrive/Colab Notebooks/grape_dataset/test'


img_size = (224, 224)
batch_size = 32
epochs = 10


train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
)

test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=img_size,
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=True
)

test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=img_size,
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False
)
```

We have to give the path of the dataset from the drive to the Google Colab we can also mention the img_size,batch_size and epochs which can be later used in the code.We can mention target_size of the image which we are going to give to the model,batch_size,class_model = categorical means encoding the categories from 0,1,2...upto number of classes

```

base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
```

```
for layer in base_model.layers[:-4]:
    layer.trainable = False
for layer in base_model.layers[-4:]:
    layer.trainable = True


x = base_model.output
x = Flatten()(x)
x = Dense(512, activation='relu')(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
x = Dense(128, activation='relu')(x)
x = Dropout(0.5)(x)
output = Dense(train_generator.num_classes, activation='softmax')(x)
```

As we are doing the transfer learning we will directly use the weights from the image net and removes the top_layer i.e dense layer and also we can observe here that we unfreezed the last 4 layer which is fine tuning,to freeze the layers completely i.e to use imagenet weight instead of training *base_ model.trainable = False* by adding False we freeze the layers to train we should write True and finally adding the dense layer for training on our required dataset

Listing 8.3: Trainig and compiling the model

```

model = Model(inputs=base_model.input, outputs=output)


optimizer = Adam(learning_rate=1e-5)
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy
    ↪ '])

plot_model(model, to_file='model_architecture.png', show_shapes=True, show_layer_names
    ↪ =True)
display(Image(filename='model_architecture.png'))



model.summary()


history = model.fit(
```

```
    train_generator,
    validation_data=test_generator,
    epochs=epochs
)




import matplotlib.pyplot as plt
import numpy as np


def show_images_with_predictions(generator, model, title):
    class_names = list(generator.class_indices.keys())
    images, true_labels = next(generator)


    preds = model.predict(images)
    pred_labels = np.argmax(preds, axis=1)
    true_label_indices = np.argmax(true_labels, axis=1)

    plt.figure(figsize=(14, 7))
    for i in range(min(8, len(images))):
        plt.subplot(2, 4, i + 1)
        plt.imshow(images[i])

        true_class = class_names[true_label_indices[i]]
        pred_class = class_names[pred_labels[i]]

        color = 'green' if true_class == pred_class else 'red'
        plt.title(f"True: {true_class}\nPred: {pred_class}", color=color, fontsize=10)
        plt.axis('off')

    plt.suptitle(title, fontsize=14)
    plt.tight_layout()
    plt.show()

show_images_with_predictions(train_generator, model, "Training Set Predictions")
show_images_with_predictions(test_generator, model, "Test Set Predictions")
```

Listing 8.4: Plotting the metrices of the model

```

import matplotlib.pyplot as plt
plt.figure(figsize=(12, 5))




plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()


plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()


test_loss, test_accuracy = model.evaluate(test_generator)
print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)

import numpy as np
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
y_pred_probs = model.predict(test_generator)
y_pred = np.argmax(y_pred_probs, axis=1)


y_true = test_generator.classes


class_labels = list(test_generator.class_indices.keys())


```

```
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=class_labels,
    ↪ yticklabels=class_labels)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()


print("\nClassification Report:\n")
print(classification_report(y_true, y_pred, target_names=class_labels))
```

Observation

Graphs

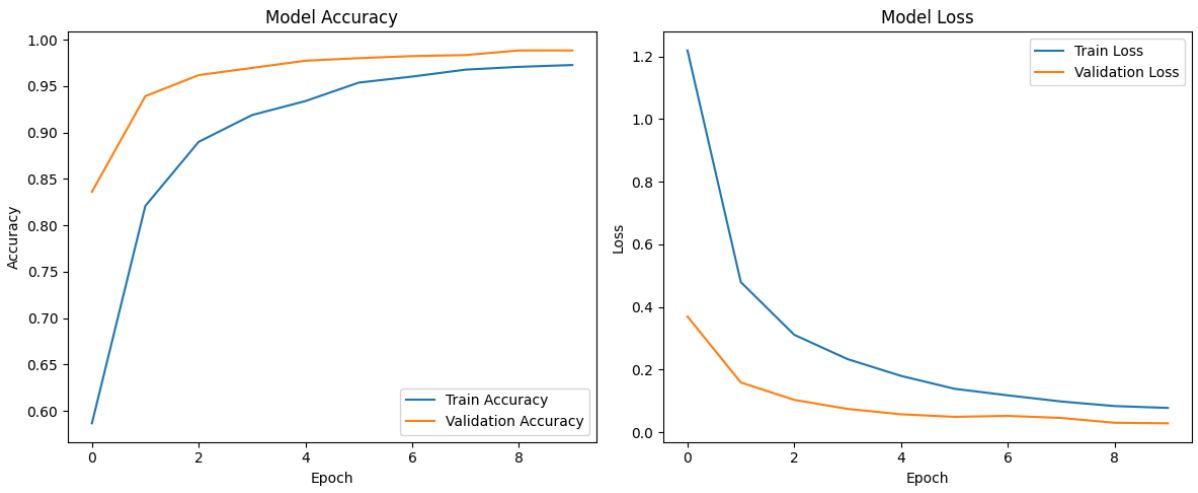


Figure 8.2: Graphs of VGG16

Confusion Matrix

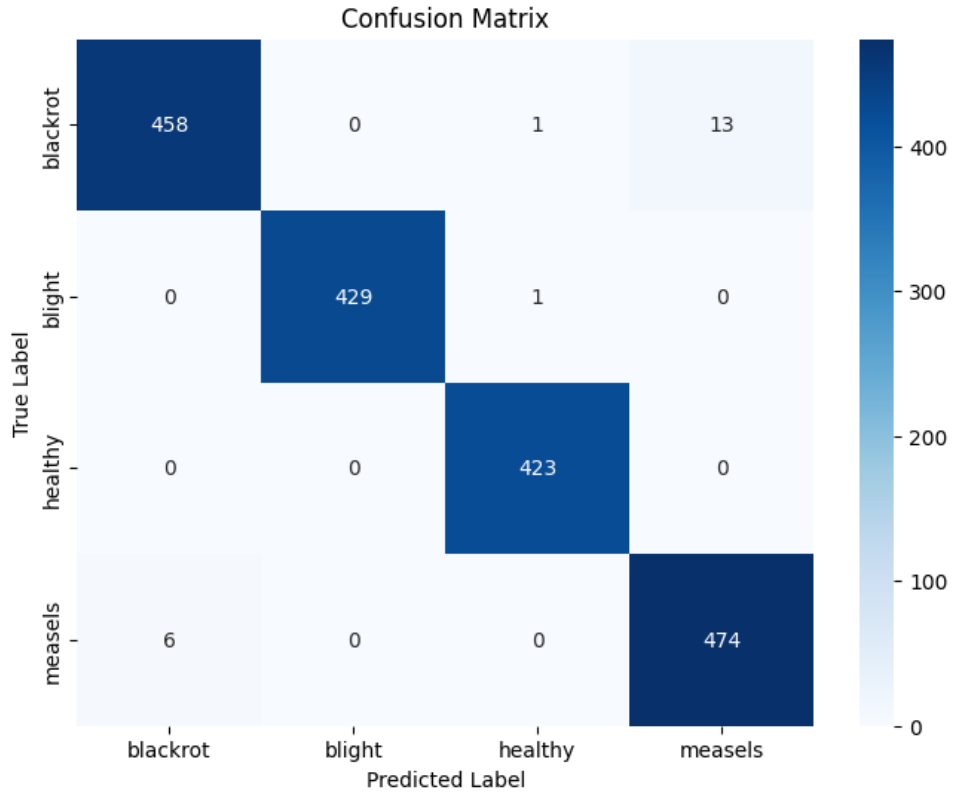


Figure 8.3: Confusion Matrix of VGG16

8.2 Classification Report

Table 8.1: Classification Report

Class	Precision	Recall	F1-Score	Support
blackrot	*0.99*	*0.97*	*0.98*	*472*
blight	*1.00*	*1.00*	*1.00*	*430*
healthy	*1.00*	*1.00*	*1.00*	*423*
measels	*0.97*	*0.99*	*0.98*	*480*
Accuracy			***0.99***	*1805*
Macro Avg	*0.99*	*0.99*	*0.99*	*1805*
Weighted Avg	*0.99*	*0.99*	*0.99*	*1805*

Result

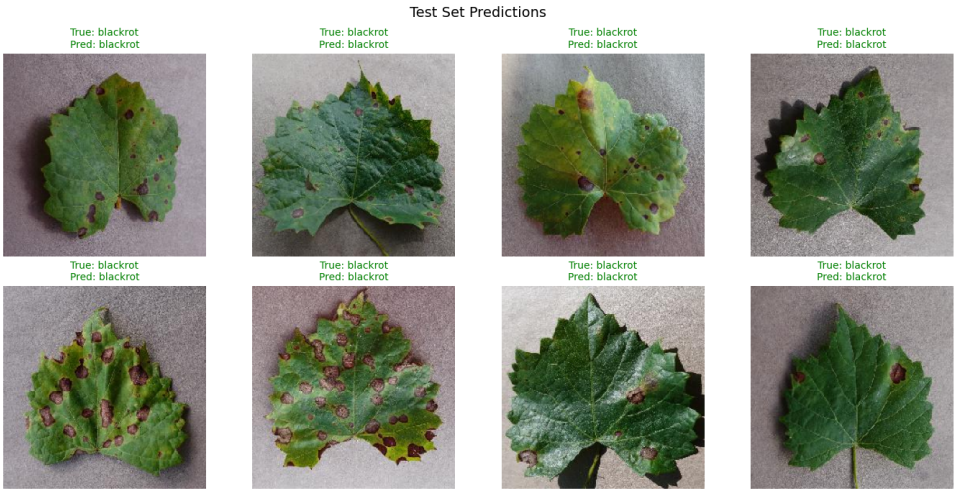


Figure 8.4: Test images of VGG16

8.3 VGG19-Architecture

8.3.1 Introduction

VGG19 is an extension of VGG16, distinguished by its greater depth,it consists of 19 layers with learnable weights, including 16 convolutional layers and 3 fully connected layers.One of the key characteristics of VGG19 is its use of small 3×3 convolution filters applied consistently across all layers, which allows the network to capture complex features while maintaining computational efficiency. It also uses ReLU activations and max pooling after groups of convolutional layers to reduce dimensionality.

While deeper and slightly more accurate than VGG16, VGG19 also requires more computation and memory, making it suitable for scenarios where accuracy is prioritized over inference speed.It performed impressively on the ImageNet dataset, achieving top-5 accuracy around 89.8%

Architecture

The VGG19 architecture is organized into five convolutional blocks followed by fully connected layers:

Block 1: 2 Conv (3×3, 64) + MaxPool
Block 2: 2 Conv (3×3, 128) + MaxPool
Block 3: 4 Conv (3×3, 256) + MaxPool
Block 4: 4 Conv (3×3, 512) + MaxPool
Block 5: 4 Conv (3×3, 512) + MaxPool
After these convolutional blocks, the model includes:

FC Layers: Flatten → FC (4096) → FC (4096) → FC (1000) → Softmax

Max pooling layers use 2×2 filters with stride 2.

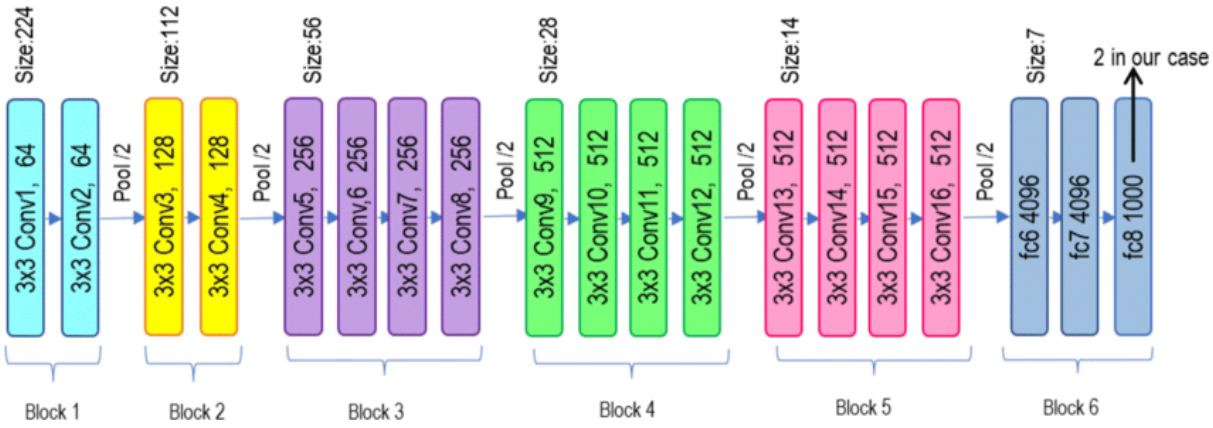


Figure 8.5: Architecture of VGG19

8.3.2 Transfer Learning Using VGG19

To use the VGG19 model we can simply import it from tensorflow inbuilt library same like VGG16.Simply we write VGG19 instead of VGG16

Listing 8.5: VGG19 Model

```
1
2 from tensorflow.keras.applications import VGG16
3
4
5 base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
6
7
8
9
10
11
12
13
```

It is our choice to freeze or unfreeze the layers using base_model.trainable

Observation for VGG19

Graphs

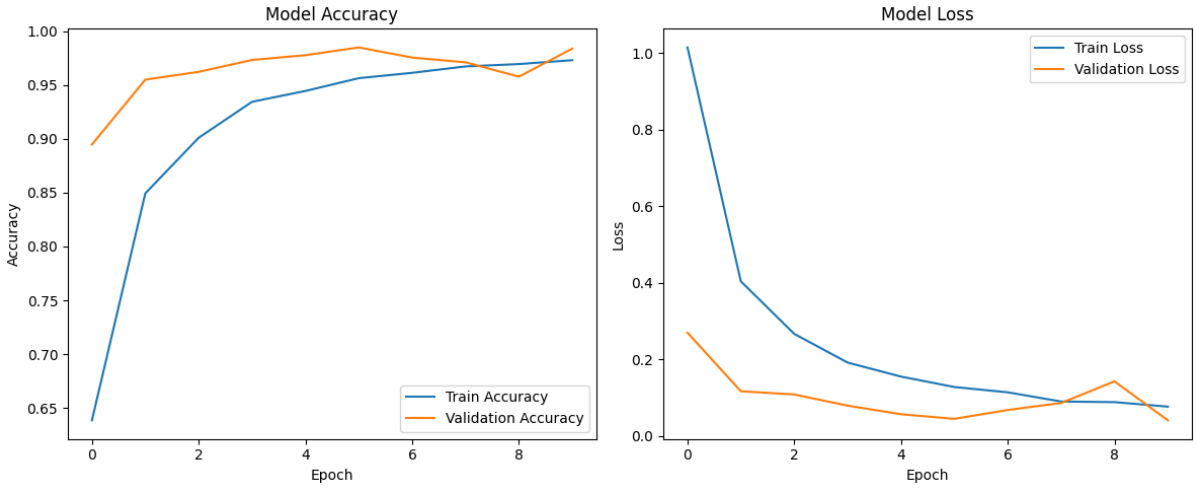


Figure 8.6: Graphs of VGG19

Confusion Matrix

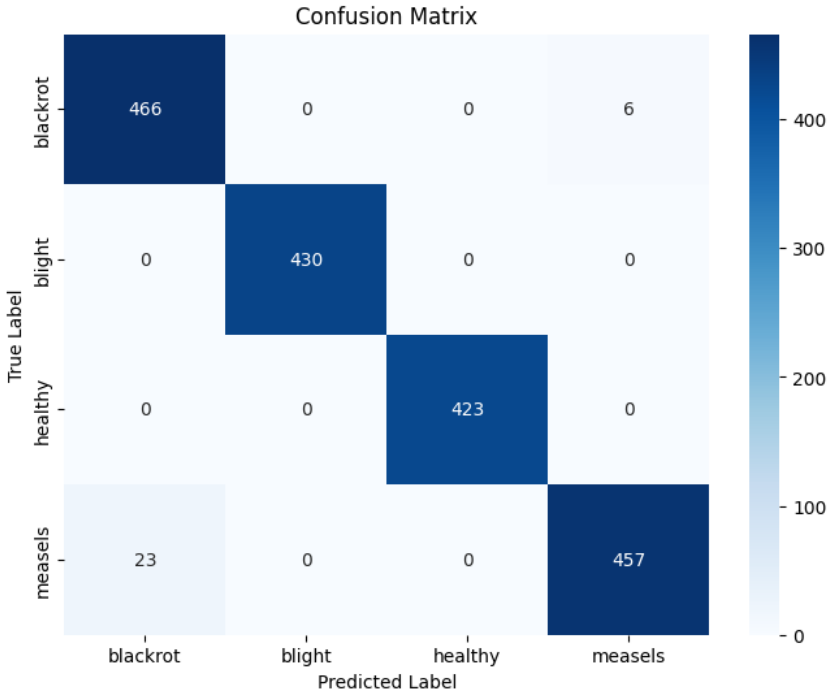


Figure 8.7: Confusion Matrix of VGG19

Classification Report

Class	Precision	Recall	F1-score	Support
Blackrot	*0.95*	*0.99*	*0.97*	*472*
Blight	*1.00*	*1.00*	*1.00*	*430*
Healthy	*1.00*	*1.00*	*1.00*	*423*
Measels	*0.99*	*0.95*	*0.97*	*480*
Accuracy			*0.98*	*1805*
Macro avg	*0.99*	*0.98*	*0.98*	*1805*
Weighted avg	*0.98*	*0.98*	*0.98*	*1805*

Table 8.2: Classification report with italicized metrics

Result

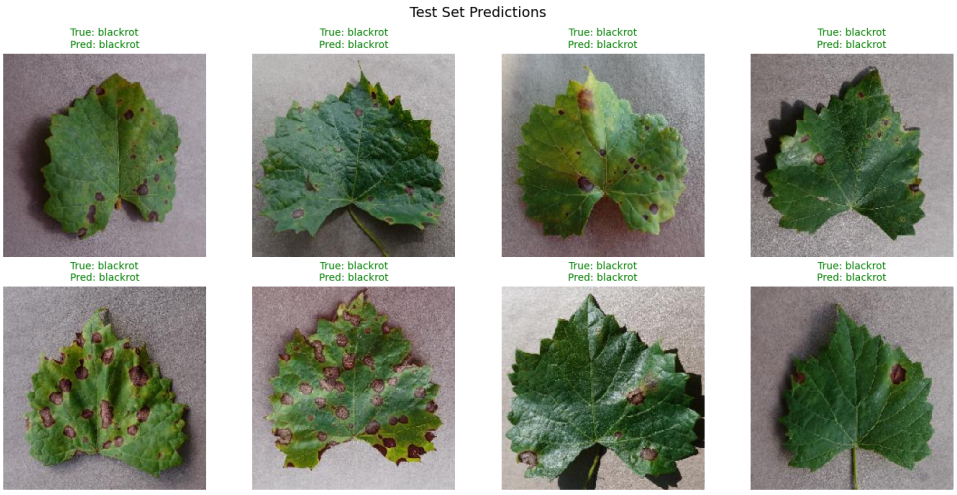


Figure 8.8: Test images of VGG19

Comparison between VGG16 and VGG19

8.4 Architecture

Feature	VGG16	VGG19
Total Layers	16 (13 conv + 3 FC)	19 (16 conv + 3 FC)
Convolutional Layers	13	16
Fully Connected (FC)	3	3
MaxPooling Layers	5	5
Conv Block Depths	2-2-3-3-3	2-2-4-4-4
Total Parameters	∼138 million	∼144 million

Table 8.3: Comparison of VGG16 and VGG19 Architectures

8.5 Performance

Aspect	VGG16	VGG19
Training Time	Faster	Slower
Inference Speed	Faster	Slower
Model Size	Smaller (∼528 MB)	Larger (∼549 MB)
Computational Cost	Lower	Higher
Overfitting Risk	Lower (slightly)	Slightly higher

Table 8.4: Performance Comparison of VGG16 and VGG19

Hyperparameters

- **Input size:** $224 \times 224 \times 3$ (RGB images)
- **Optimizer:** Often SGD with momentum (momentum = 0.9) or Adam optimizer
- **Epochs:** Typically 50–100 (depending on dataset and training)
- **Batch size:** Usually between 32 and 256 (based on hardware capacity)
- **Activation function:** ReLU (Rectified Linear Unit)
- **Learning rate:** Commonly starts around 0.01 (adjustable during training)
- **Loss function:** Cross-entropy loss (for classification tasks)
- **Dropout:** Typically 0.5 in fully connected layers (to reduce overfitting)

Conclusion

Choosing between VGG16 and VGG19, the best option depends on your priorities. VGG19 offers slightly better accuracy, but the improvement is minimal. For faster training and inference, VGG16 is the better choice due to its simpler architecture and fewer parameters. Additionally, VGG16 is easier to tune and more widely used in real-world applications, making it a practical option for most projects. Overall, if speed and simplicity are important, VGG16 is preferred, while VGG19 may be chosen when a small gain in accuracy is critical.

9. InceptionNet

9.1 InceptionNet V1 (GoogLeNet)

9.1.1 Introduction and Discovery

InceptionNet V1, popularly known as **GoogLeNet**, was proposed by **Christian Szegedy et al.** from *Google Research*. It was introduced in the research paper titled *"Going Deeper with Convolutions"*, and the architecture was submitted to the **ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) 2014**, where it achieved **state-of-the-art performance**, winning first place in classification and detection tasks.

- **Year Proposed:** 2014
- **Research Lab:** Google Research
- **Published at:** ILSVRC 2014 (part of the CVPR 2015 proceedings)

9.1.2 Motivation and Need for InceptionNet V1

Deep neural networks were becoming increasingly powerful with growing depth, but this came with significant trade-offs:

- High computational cost
- Increased number of parameters
- Risk of overfitting due to redundant layers

Architectures like **AlexNet (2012)** and **VGGNet (2014)** demonstrated that deeper networks could improve performance, but at the cost of massive computational resources.

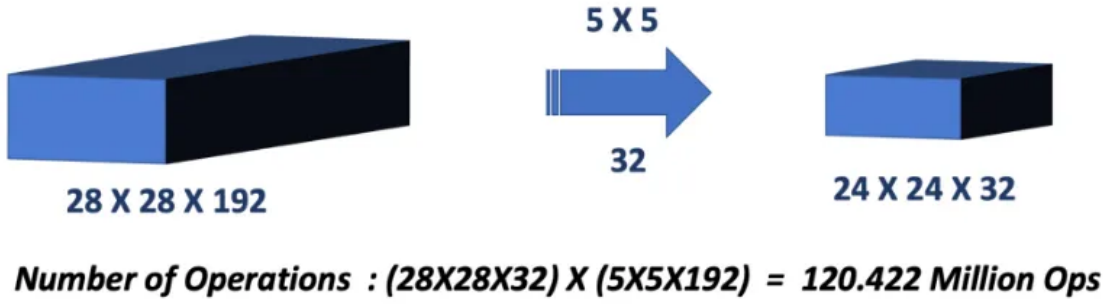
To efficiently utilize computational power while retaining or improving accuracy, InceptionNet introduced a novel module capable of **multi-scale processing** by combining convolutions of different kernel sizes and pooling operations in parallel.

9.1.3 Need of moving to Inception Net From Res Net

- InceptionNet was proposed **before ResNet** (which came in 2015).
- Its goal was to efficiently scale CNNs in both depth and width while minimizing resource usage.
- **ResNet** introduced skip connections to solve the vanishing gradient problem in very deep networks.
- **Inception** focused on extracting multi-scale features, whereas **ResNet** focused on simplifying training of deep networks.
- In practice, both are complementary and are sometimes combined (e.g., **Inception-ResNet**).

9.1.4 Dimensionality Reduction in Inception

A major innovation in the Inception module is the use of **1×1 convolutions** for dimensionality reduction. This significantly reduces the number of parameters and computational complexity.

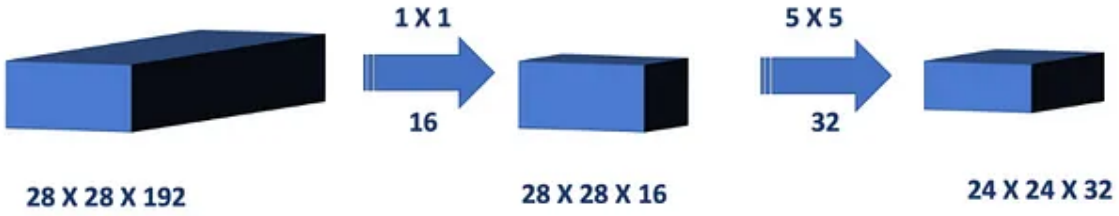


> "Adding a 1×1 convolution before a 5×5 convolution reduces the number of channels in the image when it is input to the 5×5 convolution, thus reducing both the number of parameters and the computational requirement."

This technique allowed InceptionNet to increase both depth and width without increasing computational burden.

9.1.5 Architecture of InceptionNet V1

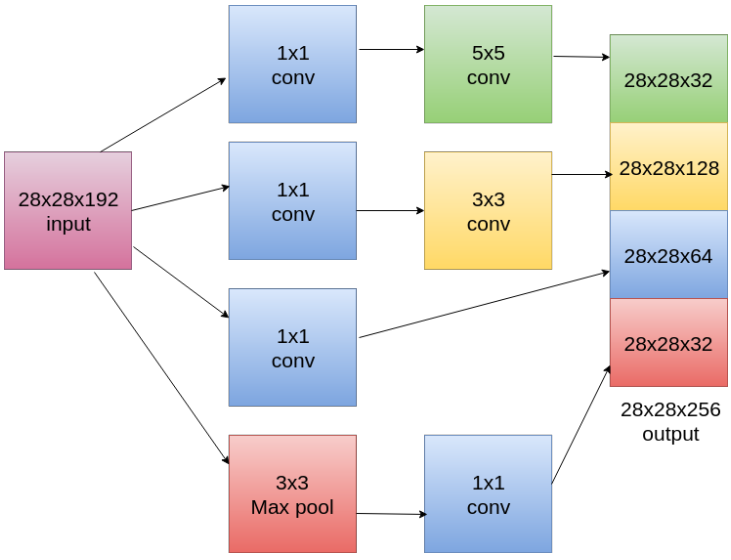
InceptionNet V1 is a **22-layer deep convolutional neural network**, composed of several *Inception modules*, the core building blocks of the architecture.



Number of Operations for 1 X 1 Conv Step : (28X28X16) X (1X1X192) = 2.4 Million Ops
Number of Operations for 5 X 5 Conv Step : (28X28X32) X (5X5X16) = 10 Million Ops
Total Number of Operations = 12.4 Million Ops

Key Design Features

- Parallel convolutional filters: **1×1, 3×3, 5×5** to extract features at multiple spatial scales.
- **1×1 convolutions** as bottleneck layers to reduce input channels to larger convolutions.
- Max pooling added in parallel to convolutional operations.
- Outputs from all branches concatenated depth-wise.

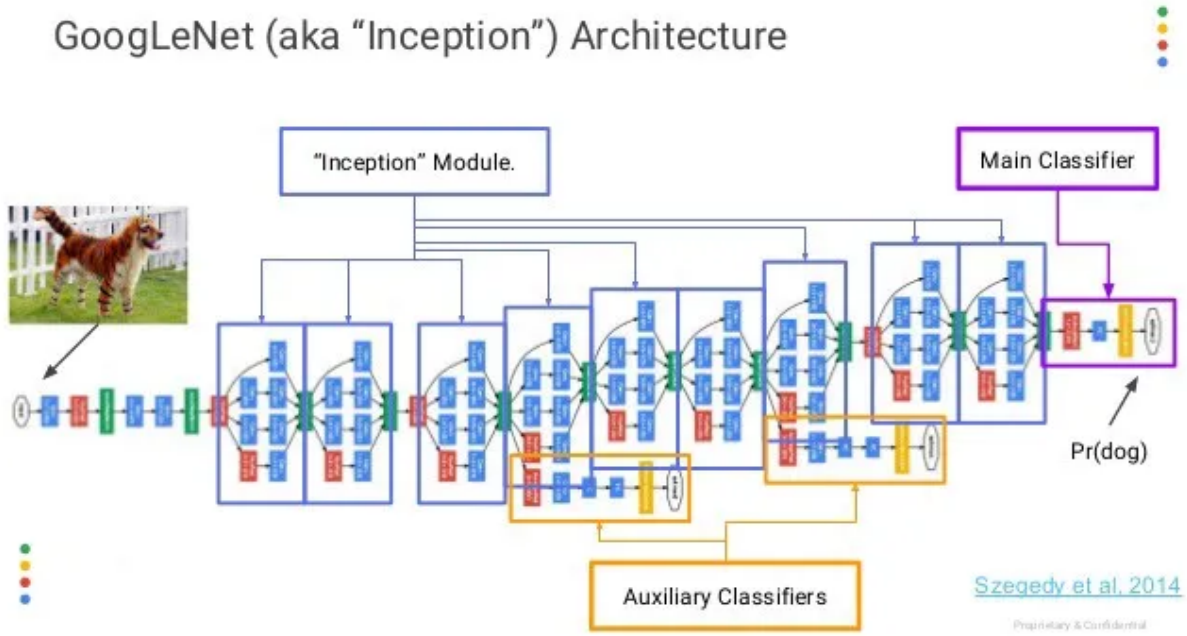


9.1.6 InceptionNet V1 – Layer Overview

- **Total layers:** 22
- **Core components:**

  - Nine Inception modules
  - Global average pooling (instead of fully connected layers)
  - Two auxiliary classifiers during training to avoid vanishing gradients

- **Number of parameters:** ∼5 million (much smaller than VGGNet)



9.1.7 Advantages of InceptionNet V1

- Reduced computational cost through 1×1 convolutions
- Efficient multi-scale feature extraction
- Lightweight and more accurate than VGGNet
- Reduced overfitting with global average pooling
- Better gradient flow with auxiliary classifiers during training

9.1.8 Applications of InceptionNet

InceptionNet V1 and its variants (V2, V3, V4) are widely used in various domains:

- Image classification
- Object detection (e.g., in YOLO, Faster R-CNN)
- Medical imaging

- Satellite image analysis
- Face recognition
- Industrial defect detection

9.2 Motivation Behind InceptionNet V2

While InceptionNet V1 (GoogLeNet) was a groundbreaking step toward efficient deep CNN architectures, it had limitations that led to further enhancements:

- High computational cost due to large convolutional filters (e.g., 5×5).
- Drastic spatial dimensionality reduction, potentially leading to information loss.
- Limited performance boost from auxiliary classifiers during early training.
- The need for deeper and wider networks with efficient resource use drove the shift to a refined version—InceptionNet V2.

9.2.1 Why Transition from Inception V1 to V2?

Problem in V1	**Solution in V2**
High cost of 5×5 convolutions	Replaced by two consecutive 3×3 convolutions
Inefficiency in computation	Factorization into 1×n followed by n×1 convolutions
Early convergence issues	Enhanced auxiliary classifiers with refined connections
Limited expressiveness	Deeper modules with better utilization of filters

9.2.2 Architectural Innovations in InceptionNet V2

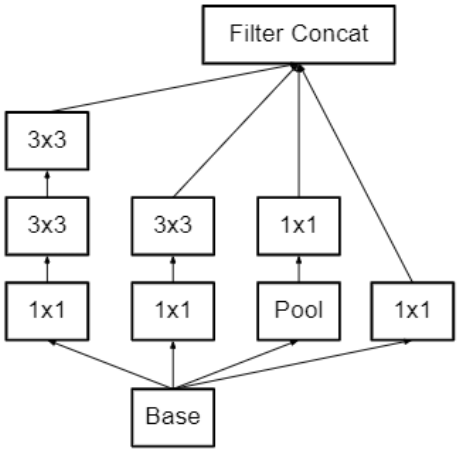
1. Factorization of Large Filters

- $5 \times 5 \rightarrow 3 \times 3 + 3 \times 3$: Maintains the same receptive field but is computationally cheaper.
- Example:

$$\text{Cost}_{5\times5} = 25 \text{ units}, \quad \text{Cost}_{3\times3\rightarrow3\times3} = 18 \text{ units}$$

2. Asymmetric Convolution Factorization

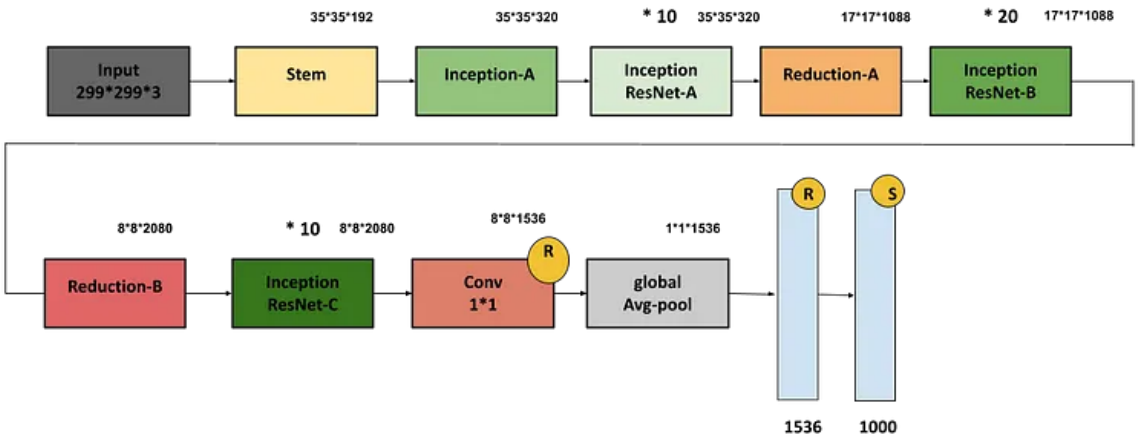
- $3 \times 3 \rightarrow 1 \times 3$ followed by 3×1: Reduces parameters by approximately 33%.
- Maintains performance while increasing model depth.



3. Refined Auxiliary Classifiers

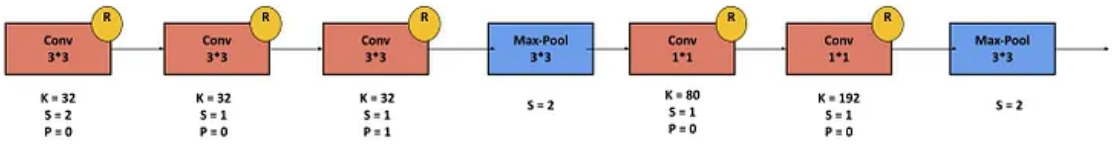
- Placed more effectively for better training support.
- Still help mitigate vanishing gradients, though not very effective in early stages.

9.2.3 Architecture Overview of InceptionNet V2

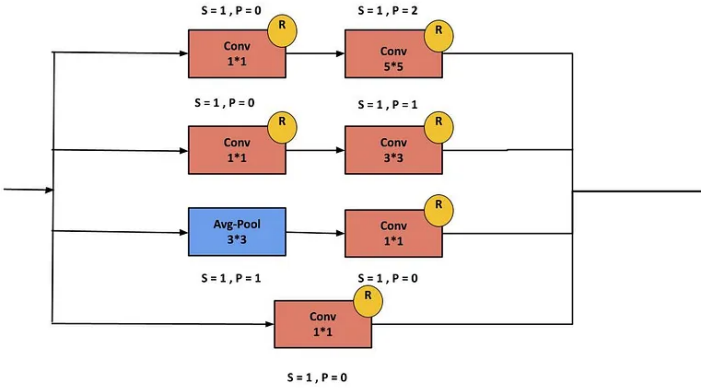


InceptionNet V2 builds upon the modular concept from V1 but with more optimized blocks. A typical flow includes:

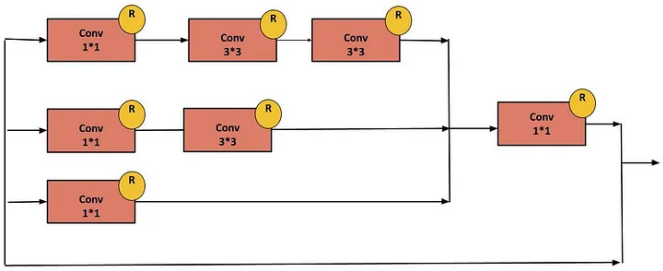
Stem block – Initial convolution and pooling layers.



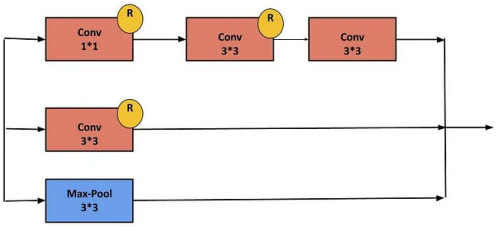
Inception-A modules – Feature extraction using small filters (1×1, 3×3).



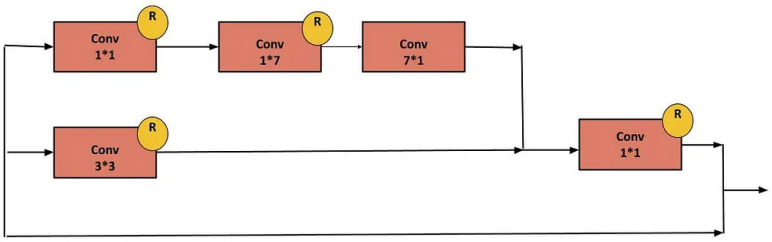
- **Inception ResNet-A Block**



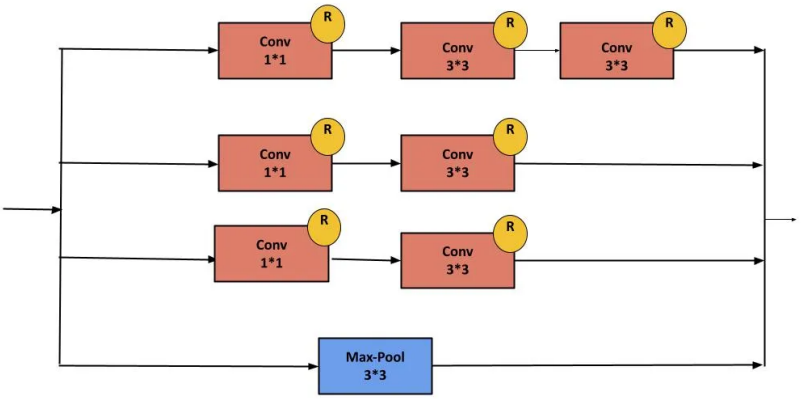
- **Reduction-A** – Efficient spatial downsampling.



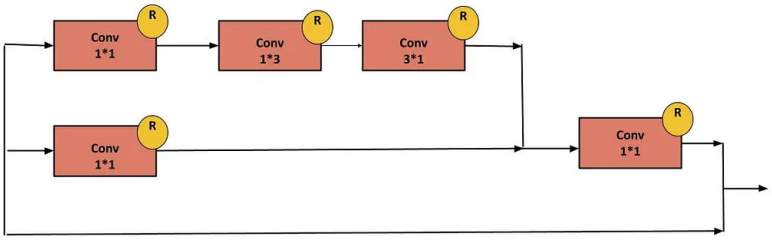
- **Inception Resnet-B module**



- **Reduction-B** – Additional downsampling.



- **Inception Resnet- C module** – Deeper and more complex feature maps.



- **Global average pooling** – Replaces fully connected layers.
- **Dropout + Softmax** – Final classification layer.
- **Note:** Each Inception module combines convolution paths (1×1, 3×3, 1×3 → 3×1, pooling) and merges them along the depth axis.

Total layers: Approximately 42
Parameter count: 11 million (still less than VGGNet)

Benefits of InceptionNet V2

- **Faster training** through factorized layers.
- **More efficient feature extraction** at multiple scales.
- **Improved convergence** and better gradient flow.
- **Reduced memory and compute cost** compared to V1 and VGGNet.
- **Modular and extensible** architecture: Basis for V3, V4, and Inception-ResNet hybrids.

Applications of InceptionNet V2

- Image classification (e.g., ImageNet).
- Object detection (e.g., Faster R-CNN, SSD).
- Medical image analysis.
- Face and gesture recognition.
- Satellite image interpretation and remote sensing.
- Real-time video analysis and surveillance.

Introduction

InceptionNet V3 is an advanced version of the Inception architecture, introduced by Christian Szegedy et al. and further refined by Google researchers. It was proposed as part of a continuous effort to increase accuracy and reduce computational cost on large-scale image recognition tasks like **ImageNet**.

- **Year Published:** 2015

- **Paper Title:** *Rethinking the Inception Architecture for Computer Vision*
- **Paper Link:** https://arxiv.org/abs/1512.00567
- **Main Goal:** To optimize performance and training speed while maintaining or improving accuracy over Inception V1/V2.

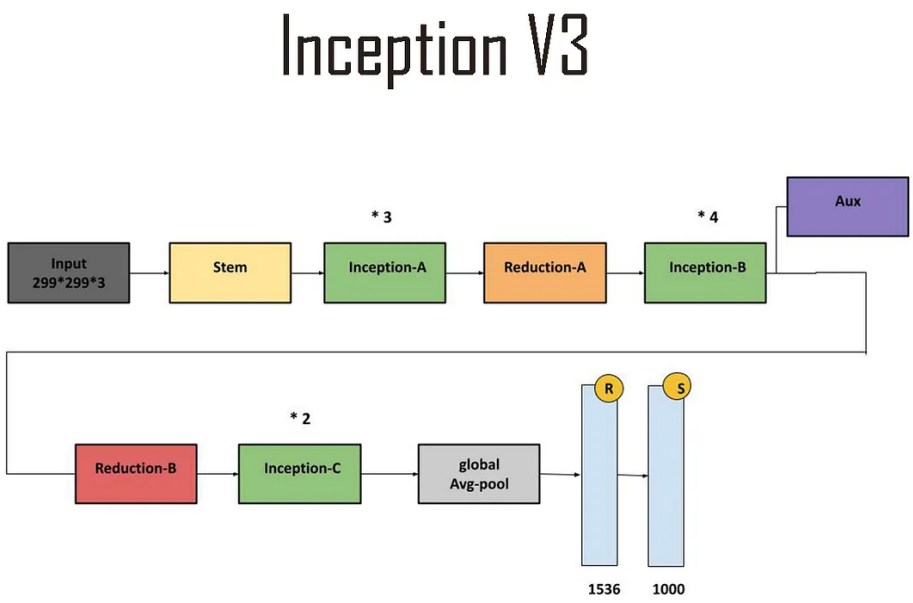
Motivations and Improvements over V2

Problem in V2	**Solution in V3**
Training speed limitations	Label smoothing,RMSProp optimizer,batch normalization
Still computationally heavy	Further factorization of convolutions
Need for deeper receptive fields	Use of larger kernel factorizations
Overfitting on large datasets	Auxiliary classifiers + regularization

9.3 Key Architectural Innovations in Inception V3

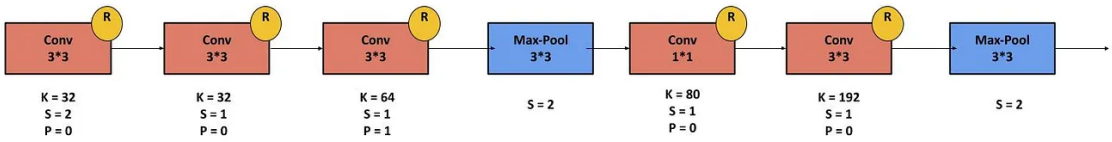
- **Factorized Convolutions:** Replacing large convolutions like 5×5 with two 3×3 convolutions. Also factorizing 3×3 into 1×3 followed by 3×1. This reduces parameters and speeds up training.
- **Efficient Grid Size Reduction:** Custom Reduction-A and Reduction-B blocks downsample efficiently without major information loss.
- **Auxiliary Classifiers:** Regularize the network and help gradients flow deeper. Placed after Inception-B modules.
- **Label Smoothing:** Prevents overconfident outputs and improves generalization.
- **Batch Normalization:** Applied after each convolution to stabilize training and improve performance.

9.3.1 Architecture Overview of InceptionNet V3

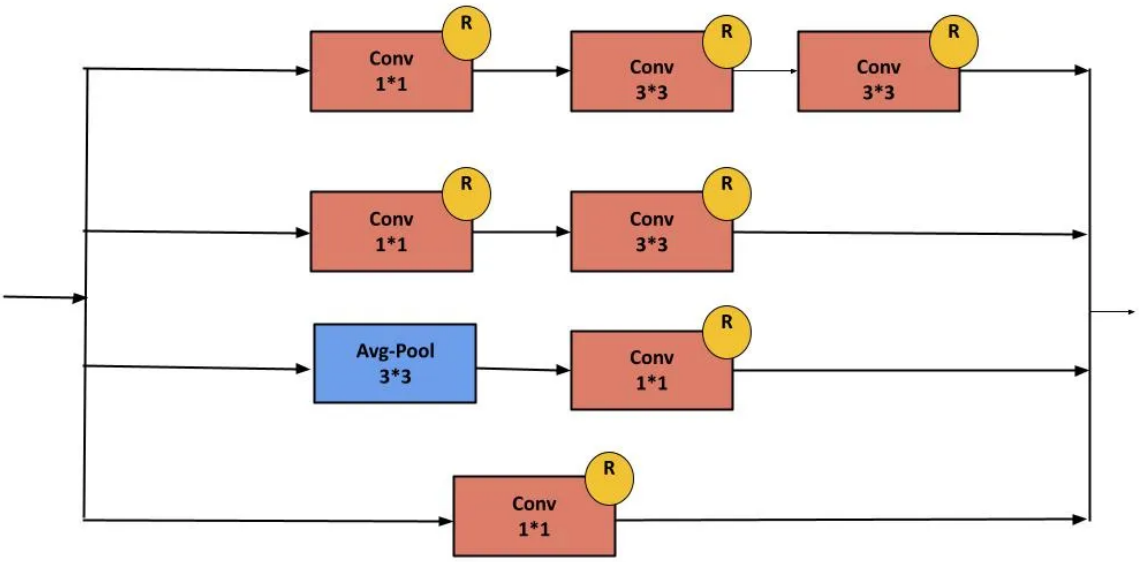


- **Input**
  → Takes a $299 \times 299 \times 3$ RGB image.

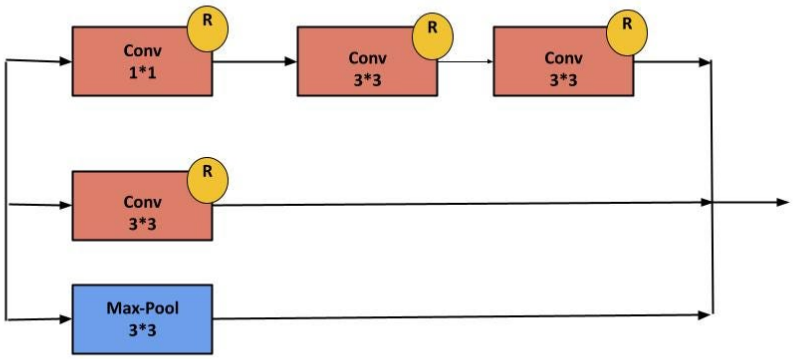
- **Stem Block**
  → Initial layers with convolution and max-pooling operations to extract low-level features.



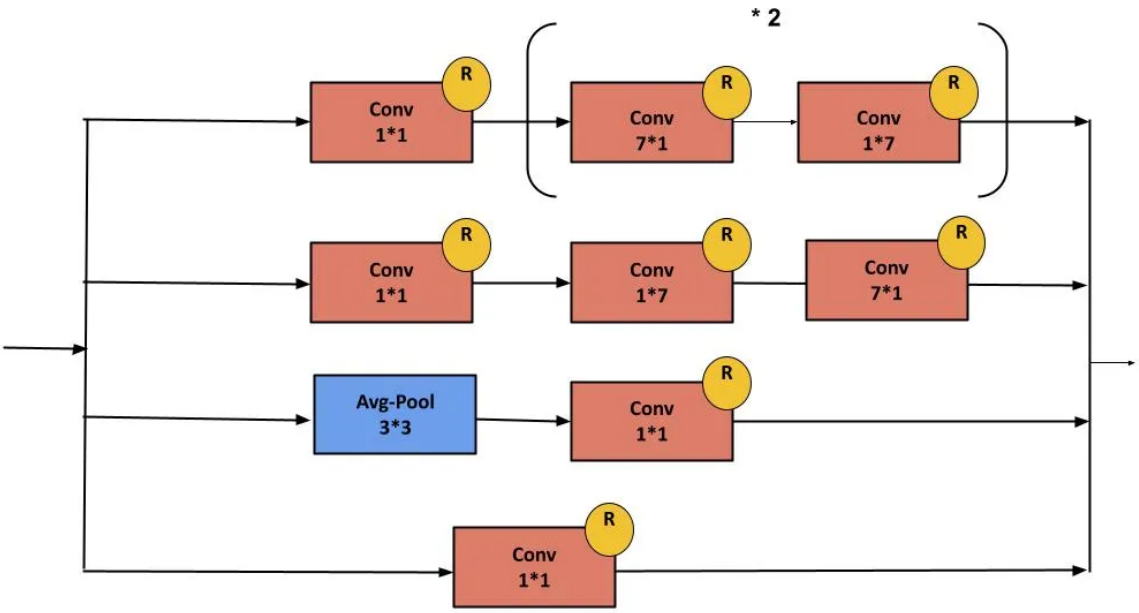
- **Inception-A $\times$ 3**
  → Three consecutive Inception-A modules using small convolutions (1×1, 3×3) for lightweight multi-path feature extraction.



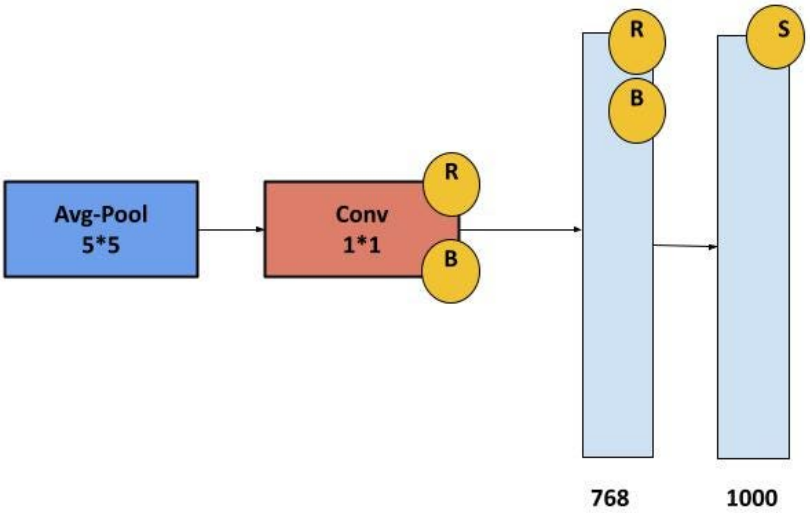
- **Reduction-A**
  → A downsampling block that reduces spatial dimensions without losing critical information.



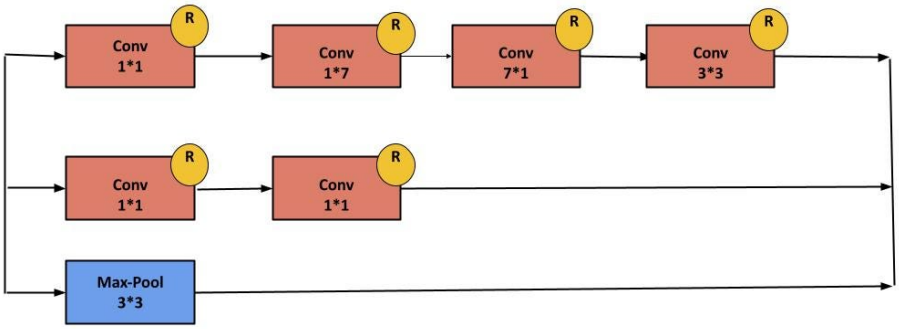
- **Inception-B $\times$ 4**
  → Four Inception-B modules with factorized convolutions (1×7 followed by 7×1) for learning deeper and more complex features efficiently.



- **Auxiliary Classifier**
  → An intermediate classifier that improves gradient flow and acts as regularization during training.

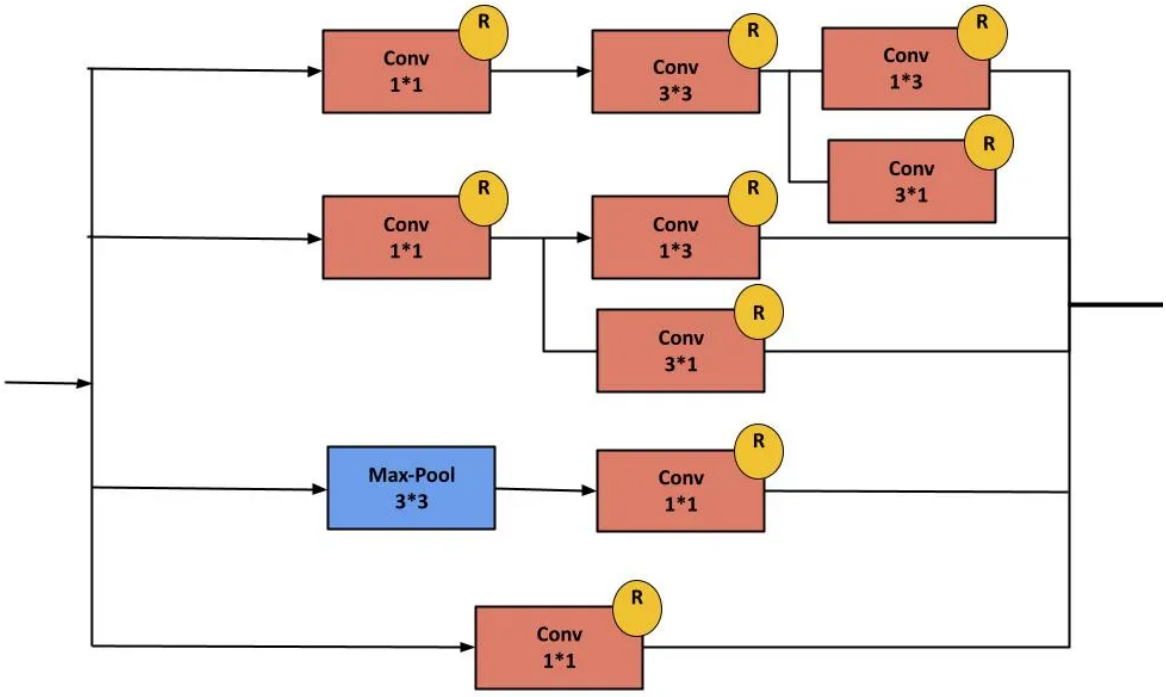


- **Reduction-B**
  → Another downsampling layer that prepares the network for deeper semantic abstraction.



- **Inception-C × 2**

  → Two Inception-C modules to extract high-level semantic features with multiple filter sizes.



- **Global Average Pooling**

  → Converts the feature maps into a $1 \times 1 \times 1536$ tensor by averaging each feature map.

- **Fully Connected Layer**
  → A dense layer that maps the 1536 feature maps to 1000 output classes (e.g., for ImageNet classification).
- **Softmax Output**
  → Produces probability scores across all classes for final prediction.

Model Summary

- **Total Layers:** 48
- **Parameters:** 23 million
- **Input Size:** 299×299
- **Output Size:** 1000 (ImageNet classes)

Benefits of InceptionNet V3

- Computationally efficient
- Higher classification accuracy on ImageNet
- Faster convergence during training
- Smart use of convolution factorization
- Effective regularization to reduce overfitting

Applications

- Large-scale image classification (ImageNet, CIFAR-100)
- Medical imaging analysis
- Industrial defect detection
- Real-time video classification
- Face and object recognition systems

10. Resnet CNN Model

Introduction

Deep learning, particularly using **Convolutional Neural Networks (CNNs)**, has led to major breakthroughs in computer vision tasks such as image classification, object detection, and segmentation. Deeper CNN architectures can learn more complex features, but increasing depth often introduces challenges like vanishing gradients, slow convergence, and model degradation.

To overcome these issues, **Residual Networks (ResNet)** were introduced by **Kaiming He et al.** in 2015. ResNet introduced the concept of *residual learning* using *skip connections*, enabling the successful training of extremely deep networks (e.g.,ResNet-18, ResNet-50, ResNet-101) with improved performance and stability.

Why Move from VGG to ResNet?

VGGNet, particularly VGG-16 and VGG-19, marked a significant advancement in convolutional neural network design by using uniform 3×3 convolution filters and increasing network depth. However, as the depth increased beyond 19 layers, several challenges became evident:

- **Degradation Problem**: Deeper VGG-like networks started to show higher training and test error, contrary to expectations.
- **Vanishing/Exploding Gradients**: Gradients became unstable during backpropagation, hindering effective learning in very deep models.
- **Computational Inefficiency**: VGG models contained millions of parameters, making them computationally expensive and hard to scale further.

To overcome these challenges by using Residual Networks (ResNet) . ResNet introduces a novel approach called residual learning, where the model learns the difference (residual) between the input and the desired output of a layer. This is mathematically represented as:

$$F(x)=H(x) - x$$
$$H(x)=F(x)+x$$

Here,

x is the input,

F(x) is the residual function, and

H(x) is the final output. The input

x is added back to the output of a few layers using a shortcut (skip) connection.

This architecture brings several advantages:

- **Improved Gradient Flow:** Skip connections allow gradients to propagate directly through the network, significantly reducing the vanishing gradient issue.
- **Scalable Depth:** ResNet models can scale to 50, 101, or even 152 layers without degradation, enabling better feature learning.
- **Higher Accuracy:** ResNet consistently outperforms VGG and other earlier architectures on large-scale benchmarks like ImageNet, while being more parameter-efficient at similar depths..

Thus, the move from VGG to ResNet was driven by the need for more scalable, efficient, and high-performing deep learning models.ResNet not only resolved the fundamental training issues of deep networks but also became the foundation for many subsequent models in deep learning.

Key Points:

- **Skip connection:** Directly adds input to output.
- **Projection shortcut:** 1×1 convolution (projection shortcut) used to match dimensions
- **Bottleneck design:** Efficient for deep networks, reduces computational load.

10.1 ResNet-18 Architecture

ResNet-18 is a deep convolutional neural network composed of 18 layers with trainable weights. The architecture includes an initial convolutional layer followed by four stages of residual blocks, ending with average pooling and a fully connected layer for classification. The image below represents the layer-wise structure of ResNet-18:

- **Input:** The input is a color image of size $224 \times 224 \times 3$.

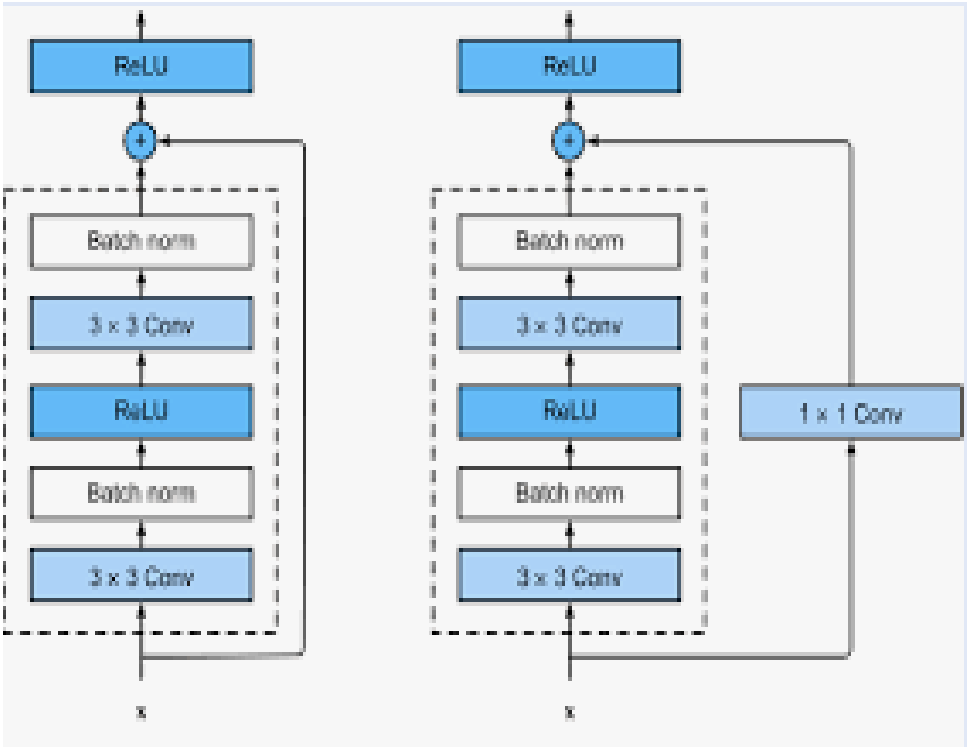


Figure 10.1: Basic Residual Block

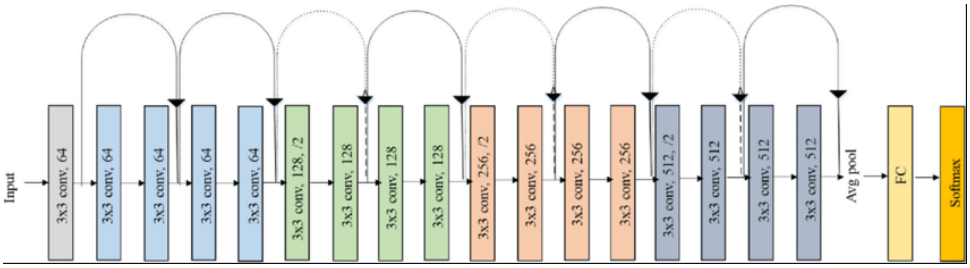


Figure 10.2: ResNet 16 Architecture diagram

- **Initial Convolution:** A 3×3 convolution layer with 64 filters extracts low-level features from the input image.

- **Conv2_x:** Two residual blocks, each with two 3×3 convolution layers with 64 filters. Skip connections are identity mappings since input and output dimensions match.

- **Conv3_x:** Two residual blocks with 128 filters. The first block uses stride 2 for downsampling and a 1×1 projection shortcut to match dimensions. The second block uses identity mapping.

- **Conv4_x:** Two residual blocks with 256 filters. The first block applies downsampling and uses a projection shortcut. The second block uses identity mapping.

- **Conv5_x:** Two residual blocks with 512 filters. The first block uses stride 2 and

a projection shortcut; the second uses an identity shortcut.

- **Average Pooling:** A global average pooling layer reduces the spatial dimension to $1 \times 1 \times 512$.
- **Fully Connected Layer:** A dense layer maps the 512-dimensional feature vector to the number of classes (e.g., 1000 for ImageNet).
- **Softmax:** Applies the softmax function to produce class probabilities.

The ResNet-18 model uses skip connections (identity or projection shortcuts) to enable efficient gradient flow and stable training, even with increased network depth. This significantly improves training performance over earlier models like VGG.

10.2 Coding Part:

Step1:Import Libraries

```
import os, random, numpy as np, matplotlib.pyplot as plt
from sklearn.metrics import classification_report, confusion_matrix,
    ↪ f1_score
import seaborn as sns
import tensorflow as tf
from tensorflow.keras.models import Model
from keras.utils import to_categorical, plot_model
from tensorflow.keras import layers, models, Input
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
import matplotlib.pyplot as plt
from IPython.display import Image
from tensorflow.keras.preprocessing.image import ImageDataGenerator,
    ↪ img_to_array, load_img, save_img
```

Step2:Data Augmentation for training

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator,
    ↪ img_to_array, load_img, save_img
from tqdm import tqdm
import shutil
# PARAMETERS
```

```
dataset_path = '/content/drive/MyDrive/chilli dataset/DS 3/train1'
target_size = 224
TARGET_IMAGES = 1100
# Data augmenter
augmenter = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)
```

```
for class_name in os.listdir(dataset_path):
    class_path = os.path.join(dataset_path, class_name)
    if not os.path.isdir(class_path):
        continue

    image_files = [f for f in os.listdir(class_path) if f.lower().endswith(('.
        ↪ jpg', '.jpeg', '.png'))]
    current_count = len(image_files)

    print(f"\nClass '{class_name}': {current_count} images")

    # CASE 1: Downsample if more than needed
    if current_count > TARGET_IMAGES:
        print(f" Too many images. Downsampling to {TARGET_IMAGES}...")
        random.shuffle(image_files)
        keep_files = image_files[:TARGET_IMAGES]
        delete_files = set(image_files) - set(keep_files)

        for file_name in delete_files:
            try:
                os.remove(os.path.join(class_path, file_name))
            except Exception as e:
                print(f"Error deleting {file_name}: {e}")
        print(f"Downsampled '{class_name}' to {TARGET_IMAGES} images.")

    # CASE 2: Augment if fewer than needed -----
    elif current_count < TARGET_IMAGES:
        needed = TARGET_IMAGES - current_count
        print(f"Generating {needed} new images...")
```

```
        i = 0
        pbar = tqdm(total=needed, desc=f"Augmenting '{class_name}'")
        while i < needed:
            for img_name in image_files:
                img_path = os.path.join(class_path, img_name)
                try:
                    img = load_img(img_path, target_size=(target_size, target_size
                        ↪ ))
                    x = img_to_array(img)
                    x = np.expand_dims(x, axis=0)

                    aug_iter = augmenter.flow(x, batch_size=1)
                    aug_img = next(aug_iter)[0].astype(np.uint8)
```

```
                    save_name = f"aug_{i}_{img_name}"
                    save_path = os.path.join(class_path, save_name)
                    save_img(save_path, aug_img)

                    i += 1
                    pbar.update(1)

                    if i >= needed:
                        break
                except Exception as e:
                    print(f"Error processing {img_name}: {e}")
                    continue
        pbar.close()
        print(f" Augmentation complete for class '{class_name}'.")

    #CASE 3: Already balanced
    else:
        print("Already has exactly 1100 images. Skipping.")
```

Output:

```
Class 'micronutrient deficiency': 1100 images
Already has exactly 1100 images. Skipping.
```

```
 Class 'healthy': 1100 images
 Already has exactly 1100 images. Skipping.

 Class 'Bacterial leaf spot': 1100 images
Already has exactly 1100 images. Skipping.

 Class 'up curl': 1100 images
 Already has exactly 1100 images. Skipping.

 Class 'cercospora leaf spot': 1100 images
Already has exactly 1100 images. Skipping.

 Class 'leaf spot': 1100 images
Already has exactly 1100 images. Skipping
```

Note: To perform augmentation on the testing dataset, simply change the **dataset_path** variable to the path of the test folder and run the same script with reqiured **Target Images**.

Step3:Data Generator Setup

```
# Image settings
img_size = 224 # or your preferred size
batch_size = 40
train_dir = '/content/drive/MyDrive/chilli dataset/DS 3/train1'
test_dir='/content/drive/MyDrive/chilli dataset/DS 3/test1'

# Create training and validation ImageDataGenerators
train_datagen = ImageDataGenerator(
    rescale=1./255,
    validation_split=0.2
)

test_datagen = ImageDataGenerator(rescale=1./255)

# Training generator
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(img_size, img_size),
```

```
    batch_size=batch_size,
    color_mode='rgb',
    class_mode='categorical',
    subset='training' # Subset for training
)

# Validation generator
validation_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(img_size, img_size),
    batch_size=batch_size,
    color_mode='rgb',
    class_mode='categorical',
    subset='validation' # Subset for validation
)

# Testing generator
test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=(img_size, img_size),
    batch_size=batch_size,
    color_mode='rgb',
    class_mode='categorical'
)
```

```
Found 5280 images belonging to 6 classes.
Found 1320 images belonging to 6 classes.
Found 1498 images belonging to 6 classes.
```

Step4:Visualization of Sample Images from Each Class

To visually inspect the dataset, one random image from each of the six classes is selected and displayed in a 2×3 grid using Matplotlib.

```
import random
# Automatically get class folder names
class_names= ['Bacterial leaf spot', 'cercospora leaf spot', 'healthy', 'leaf
    ↪ spot', 'micronutrient deficiency', 'up curl']

# Plot one random image from each class folder
plt.figure(figsize=(15, 8))
```

```
for i, class_name in enumerate(class_names):
    class_path = os.path.join(train_dir, class_name)
    image_files = [f for f in os.listdir(class_path) if f.lower().endswith(('png
        ↪ ', 'jpg', 'jpeg'))]

    if not image_files:
        print(f"No images found in {class_name}")
        continue

    # Randomly select one image
    selected_image = random.choice(image_files)
    img_path = os.path.join(class_path, selected_image)
    img = load_img(img_path, target_size=(IMAGE_SIZE))

    plt.subplot(2, 3, i + 1)
    plt.imshow(img)
    plt.title(f"Class: {class_name}")
    plt.axis('off')

plt.tight_layout()
plt.show()
```

Step5:Class Distribution Visualization

This code counts the number of images in each class for both training and testing datasets. It then visualizes the image distribution using pie charts and prints the class-wise image counts.

```
# Function to count images in each class
def count_images(dataset_path):
    class_counts = {}
    for label in os.listdir(dataset_path):
        label_path = os.path.join(dataset_path, label)
        if os.path.isdir(label_path):
            num_images = len(os.listdir(label_path))
            class_counts[label] = num_images
    return class_counts

# Count images in training and testing datasets
train_counts = count_images(train_dir)
test_counts = count_images(test_dir)
```

```
# Plot pie charts
def plot_pie_chart(class_counts, title):
    labels = list(class_counts.keys())
    sizes = list(class_counts.values())
    plt.figure(figsize=(6, 6))
    plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=0, colors=['
        ↪ skyblue', 'lightgreen', 'lightcoral', 'gold'])
    plt.title(title)
    plt.show()

# Plot for training data
plot_pie_chart(train_counts, "Training Data Distribution")

# Plot for testing data
plot_pie_chart(test_counts, "Testing Data Distribution")

# Print the number of images in each class
print("Training Data Counts:")
for label, count in train_counts.items():
    print(f"{label}: {count} images")

print("\nTesting Data Counts:")
for label, count in test_counts.items():
    print(f"{label}: {count} images")
```

Step 6: Improved ResNet18 Architecture with Dropout

The following Python code defines an improved version of the ResNet18 architecture using TensorFlow and Keras. This model includes **Dropout layers** and **L2 regularization** to reduce overfitting and improve generalization:

```
from tensorflow.keras import regularizers

def conv_block(x, filters, kernel_size=3, stride=1, downsample=False):
    identity = x
    y = layers.Conv2D(filters, kernel_size, strides=stride, padding='same',
                      kernel_initializer='he_normal', kernel_regularizer=
                          ↪ regularizers.l2(1e-4))(x)
    y = layers.BatchNormalization()(y)
    y = layers.ReLU()(y)
    y = layers.Conv2D(filters, kernel_size, strides=1, padding='same',
```

```
                    kernel_initializer='he_normal', kernel_regularizer=
                        ↪ regularizers.l2(1e-4))(y)
    y = layers.BatchNormalization()(y)
    if downsample or x.shape[-1] != filters:
        identity = layers.Conv2D(filters, 1, strides=stride, padding='same',
                                 kernel_initializer='he_normal', kernel_regularizer
                                    ↪ =regularizers.l2(1e-4))(identity)
        identity = layers.BatchNormalization()(identity)
    y = layers.Add()([y, identity])
    y = layers.ReLU()(y)
    y = layers.Dropout(0.3)(y) # Dropout added
    return y

def build_resnet18(input_shape=(224, 224, 3), num_classes=3):
    inputs = Input(shape=input_shape)
    x = layers.Conv2D(64, 7, strides=2, padding='same', kernel_regularizer=
        ↪ regularizers.l2(1e-4))(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)
    x = layers.MaxPooling2D(3, strides=2, padding='same')(x)

    x = conv_block(x, 64)
    x = conv_block(x, 64)
    x = conv_block(x, 128, stride=2, downsample=True)
    x = conv_block(x, 128)
    x = conv_block(x, 256, stride=2, downsample=True)
    x = conv_block(x, 256)
    x = conv_block(x, 512, stride=2, downsample=True)
    x = conv_block(x, 512)
    x = layers.GlobalAveragePooling2D()(x)
```

```
    x = layers.Dropout(0.5)(x) # Dropout before classification
    outputs = layers.Dense(num_classes, activation='softmax')(x)
    return models.Model(inputs, outputs)

model = build_resnet18(input_shape=(224, 224, 3), num_classes=num_classes)
```

Explanation:

- The `conv_block` function defines a residual block with optional downsampling and Dropout (0.3) to enhance generalization.
- The `build_resnet18` function builds the overall architecture:

  - Starts with an initial convolution, batch normalization, and max pooling.
  - Applies four stages of residual blocks with increasing filter sizes: 64, 128, 256, and 512.
  - Uses `GlobalAveragePooling2D`, a final Dropout layer (0.5), and a softmax Dense layer for classification.

- L2 regularization is used in convolution layers to reduce overfitting.

Step 7: Model Compilation

To improve training performance and generalization, the model is compiled with a dynamic learning rate and label smoothing. The following code demonstrates this:

```
initial_lr = 0.001

def scheduler(epoch, lr):
    if epoch > 5:
        return lr * 0.5
    return lr

lr_callback = tf.keras.callbacks.LearningRateScheduler(scheduler)

opt = tf.keras.optimizers.Adam(learning_rate=initial_lr)
```

```
# Label smoothing = 0.1 to reduce overconfidence
model.compile(optimizer=opt,
              loss=tf.keras.losses.CategoricalCrossentropy(label_smoothing=0.1),
              metrics=['accuracy'])

model.summary()
```

Explanation:

- The `scheduler` function gradually decreases the learning rate by half after 5 epochs to allow finer updates during later training.
- `LearningRateScheduler` is used to apply this schedule during model training.
- The model uses the `Adam` optimizer initialized with a learning rate of 0.001.
- `CategoricalCrossentropy` with **label smoothing** (value = 0.1) prevents the model from becoming overconfident in its predictions, helping to improve generalization.

- The model is compiled with `accuracy` as the evaluation metric.

ms1.png

Figure 10.3: model summary

Step8:Model Training

```
history = model.fit(
    train_generator,
    epochs=20,
    validation_data=validation_generator,

)
```

Explanation: The model was trained using the `model.fit()` function:

- **train_generator:** Provides augmented training images in batches.
- **epochs = 20:** The model iterates over the entire training data for 20 cycles.
- **validation_data:** Used to evaluate performance after each epoch.

The training history is stored in the variable `history`, which can be used to plot learning curves.

Step 9:Model Architecture Visualization:

This code generates and displays a block diagram (architecture visualization) of your model using plot model() from tensorflow.keras.utils.

```
plot_model(model, to_file='chili_leaf_reset18_model.png', show_shapes=True,
    ↪ show_layer_names=True, rankdir='LR')

# Display block diagram
try:
    display(Image(filename='chili_leaf_resnet18_model.png'))
except NameError: # Catch NameError if Image is not defined
    print("Model plot saved as 'chili_leaf_resnet18_model.png'")
```

Step 10:Plot the Graphs

This code plots the training and validation accuracy and loss curves to visualize the model's performance over epochs.

```
# Plot training history
plt.figure(figsize=(12,5))
# Accuracy plot
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
# Loss plot
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.tight_layout()
plt.show()
```

Output:

resnet18grap.png

Step 11:Print Values

```
# 1. Print Final Training Accuracy and Loss
final_train_accuracy = history.history['accuracy'][-1]
final_train_loss = history.history['loss'][-1]
print(f"Final Training Accuracy: {final_train_accuracy * 100:.2f}%")
print(f"Final Training Loss: {final_train_loss:.4f}")

# 2. Print Final Validation Accuracy and Loss
final_val_accuracy = history.history['val_accuracy'][-1]
final_val_loss = history.history['val_loss'][-1]
print(f" Final Validation Accuracy: {final_val_accuracy * 100:.2f}%")
print(f" Final Validation Loss: {final_val_loss:.4f}")

# 3. Evaluate and print Test Accuracy and Loss
test_loss, test_accuracy = model.evaluate(test_generator, verbose=0)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
print(f"Test Loss: {test_loss:.4f}")
```

Output:

```
Final Training Accuracy: 96.61%
```

```
Final Training Loss: 0.6580
Final Validation Accuracy: 94.24%
Final Validation Loss: 0.7089
Test Accuracy: 96.93%
Test Loss: 0.6793
```

Step 11: Model Evaluation with Confusion Matrix and Classification Report

The following code evaluates the performance of the trained model on the validation dataset. It generates a confusion matrix and a classification report:

- **Confusion Matrix:** Shows the number of correct and incorrect predictions for each class.
- **Classification Report:** Provides precision, recall, F1-score, and support for each class.

```
# Ensure validation_generator is not shuffled
# If not already done earlier, re-define it here with shuffle=False
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(img_size, img_size),
    batch_size=batch_size,
    color_mode='rgb',
    class_mode='categorical',
    subset='training' # Subset for training
)
# Validation generator
validation_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(img_size, img_size),
    batch_size=batch_size,
    color_mode='rgb',
    class_mode='categorical',
    subset='validation' ,
    shuffle=False
)
# Get true labels from validation set
Y_true = validation_generator.classes
```

```
# --- Predict class probabilities using the trained model
# Cast the result of np.ceil to an integer for the 'steps' argument
Y_pred_probs = model.predict(validation_generator, steps=int(np.ceil(
    ↪ validation_generator.samples / validation_generator.batch_size)))
Y_pred = np.argmax(Y_pred_probs, axis=1)
class_names = list(validation_generator.class_indices.keys())
cm = confusion_matrix(Y_true, Y_pred)
# --- Plot the confusion matrix ---
plt.figure(figsize=(10, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
           xticklabels=class_names, yticklabels=class_names)
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.xticks(rotation=45)
plt.yticks(rotation=0)
plt.tight_layout()
plt.show()
# --- Print classification report ---
print("\nClassification Report:")
print(classification_report(Y_true, Y_pred, target_names=class_names))
```

Output:

Step 12:Save the Model

```

#Google Colab
model.save('//content/drive/MyDrive/trained _models/chilli_leaf_resnet18_model.
    ↪ h5')

#Kaggle
# 1. Create directory for saving model (optional)
save_dir = '/kaggle/working/'
os.makedirs(save_dir, exist_ok=True)
# 2. Save the model
model_path = os.path.join(save_dir, 'chili_leaf_disease_resnet18_model.h5')
model.save(model_path)
print(f"\nModel saved at '{model_path}'")
```

rs18CM.png

Step 13:Test the Model

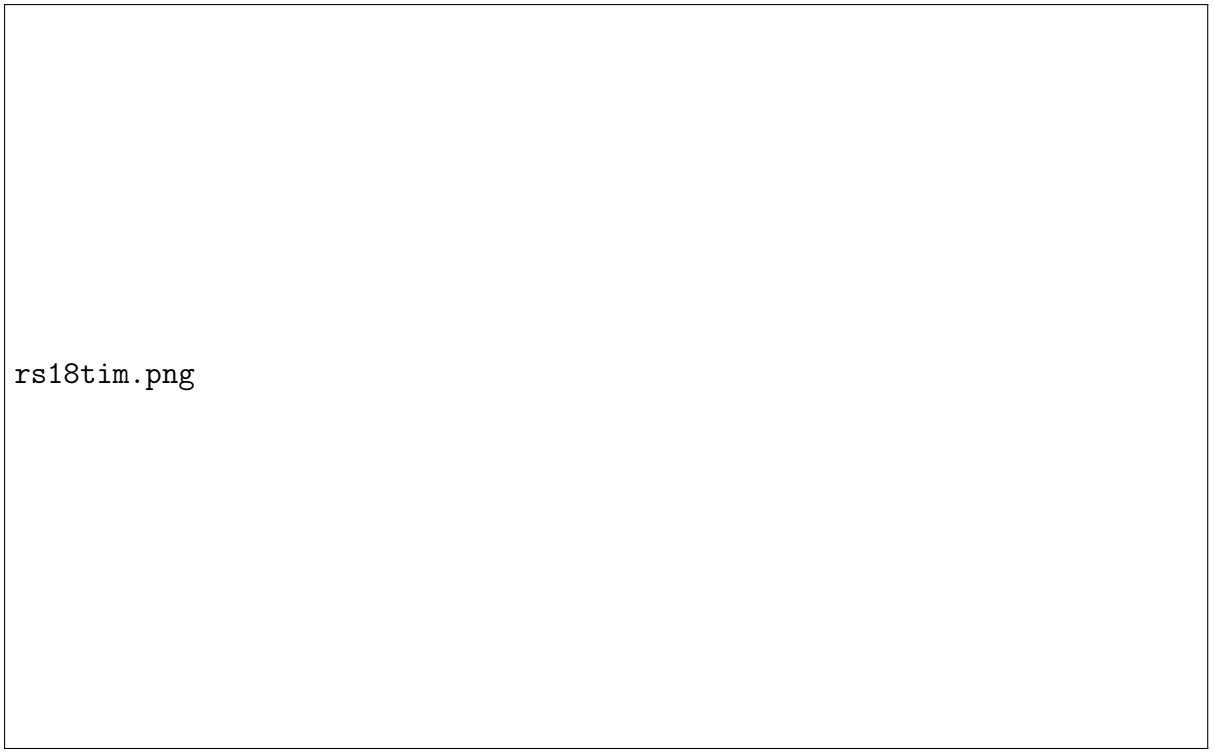
This code randomly selects 10 test images, predicts their classes using the trained model, and displays them with their true and predicted labels for visual comparison

```
# Load trained model
model = load_model('//content/drive/MyDrive/trained _models/
    ↪ chilli_leaf_resnet18_model.h5')
# Set test folder path
test_folder = '/content/drive/MyDrive/chilli dataset/DS 3/test1/'
# Get class labels from test folder
class_labels = sorted(os.listdir(test_folder))
print("Class Labels:", class_labels)
# Collect all image paths and their true labels
all_images = []
for class_name in class_labels:
    class_path = os.path.join(test_folder, class_name)
    if os.path.isdir(class_path):
        for fname in os.listdir(class_path):
            if fname.lower().endswith(('.jpg', '.jpeg', '.png')):
                full_path = os.path.join(class_path, fname)
# Shuffle and choose 10 random images
random.shuffle(all_images)
sample_images = all_images[:10]
```

rs18CR.png

```
all_images.append((full_path, class_name))
```

```
# Predict and display
plt.figure(figsize=(20, 5))
for i, (img_path, true_label) in enumerate(sample_images):
    img = image.load_img(img_path, target_size=(224, 224))
    img_array = image.img_to_array(img) / 255.0
    img_array = np.expand_dims(img_array, axis=0)
    prediction = model.predict(img_array, verbose=0)
    predicted_class = np.argmax(prediction, axis=1)[0]
    predicted_label = class_labels[predicted_class]
    plt.subplot(2, 5, i+1)
    plt.imshow(img)
    plt.title(f"Pred: {predicted_label}\nTrue: {true_label}", fontsize=10)
    plt.axis('off')
plt.tight_layout()
plt.show()
```



10.3 ResNet 50 Architecture

Introduction

ResNet-50 is a deep convolutional neural network consisting of 50 layers, designed to improve training efficiency and accuracy in deep models. It is part of the Residual Network (ResNet) family, which introduces **residual connections**—shortcuts that skip one or more layers—to address the vanishing gradient problem common in deep networks.

Unlike simpler ResNet models like ResNet-18 or ResNet-34, which use 2-layer residual blocks, ResNet-50 employs a **bottleneck structure** comprising three layers: 1×1, 3×3, and another 1×1 convolution.

ResNet-50 offers a good trade-off between depth and computational cost, making it highly effective for image classification tasks and widely used in transfer learning applications.

Why Use ResNet-50?

ResNet-50 offers several advantages over shallower architectures like ResNet-18:

- **Higher Accuracy:** Due to its deeper architecture and use of bottleneck blocks, ResNet-50 can learn more complex and abstract features, improving overall classification performance.

- **Better Generalization:** With its increased depth, ResNet-50 generalizes better on large and fine-grained datasets, especially where subtle differences between classes need to be captured.
- **Transfer Learning Friendly:** ResNet-50 is widely adopted in transfer learning applications, with many pre-trained models available that can be fine-tuned for specific tasks, saving training time and improving results.

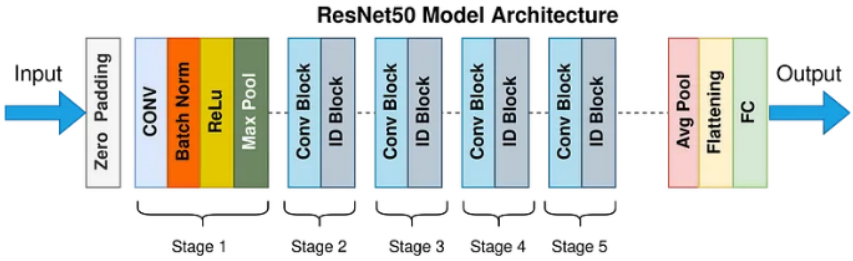


Figure 10.4: ResNet 50 Architecture diagram

The ResNet-50 architecture is a deep convolutional neural network that incorporates residual learning to enable the training of very deep networks by using identity shortcuts. The model can be broken into the following stages:

- **Input Layer:** Accepts RGB images, typically of size $224 \times 224 \times 3$.
- **Stage 1: Initial Convolution and Pooling**
  - Zero Padding
  - 7x7 Convolution with 64 filters, stride 2
  - Batch Normalization
  - ReLU Activation
  - 3x3 Max Pooling, stride 2
- **Stages 2 to 5: Residual Blocks with Skip Connections**
  - **Stage 2:** 1 Convolution Block + 2 Identity Blocks
  - **Stage 3:** 1 Convolution Block + 3 Identity Blocks
  - **Stage 4:** 1 Convolution Block + 5 Identity Blocks
  - **Stage 5:** 1 Convolution Block + 2 Identity Blocks
- **Final Layers:**

  - Global Average Pooling
  - Flattening
  - Fully Connected (Dense) Layer with Softmax Activation

- **Output Layer:** Produces class probabilities.

ResNet-50 allows deeper feature learning with improved accuracy, especially suitable for large-scale and fine-grained image classification tasks.

Transfer Learning Using ResNet 50

To use the ResNet 50 model we can simply import it from tensorflow inbuilt library .

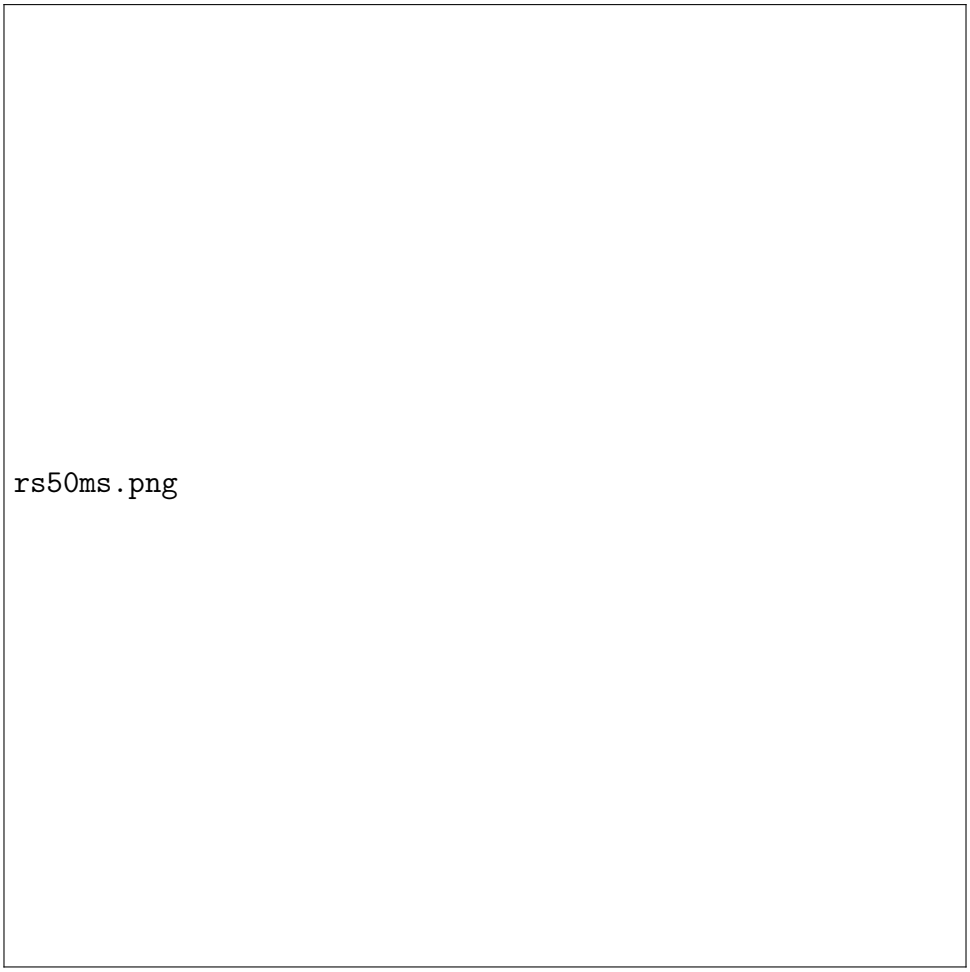
```
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.models import Model
from tensorflow.keras.layers import GlobalAveragePooling2D, Dropout, Dense,
    ↪ Flatten

def build_resnet50(input_shape=(224, 224, 3), num_classes=6):
    base_model = ResNet50(include_top=False, weights='imagenet', input_shape=
        ↪ input_shape)
    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = Flatten()(x) # Flatten layer added here
    x = Dropout(0.3)(x) # Dropout for regularization
    outputs = Dense(num_classes, activation='softmax')(x)
    model = Model(inputs=base_model.input, outputs=outputs)
    return model

# Build the model
model = build_resnet50(input_shape=(224, 224, 3), num_classes=num_classes)
```

Observations:

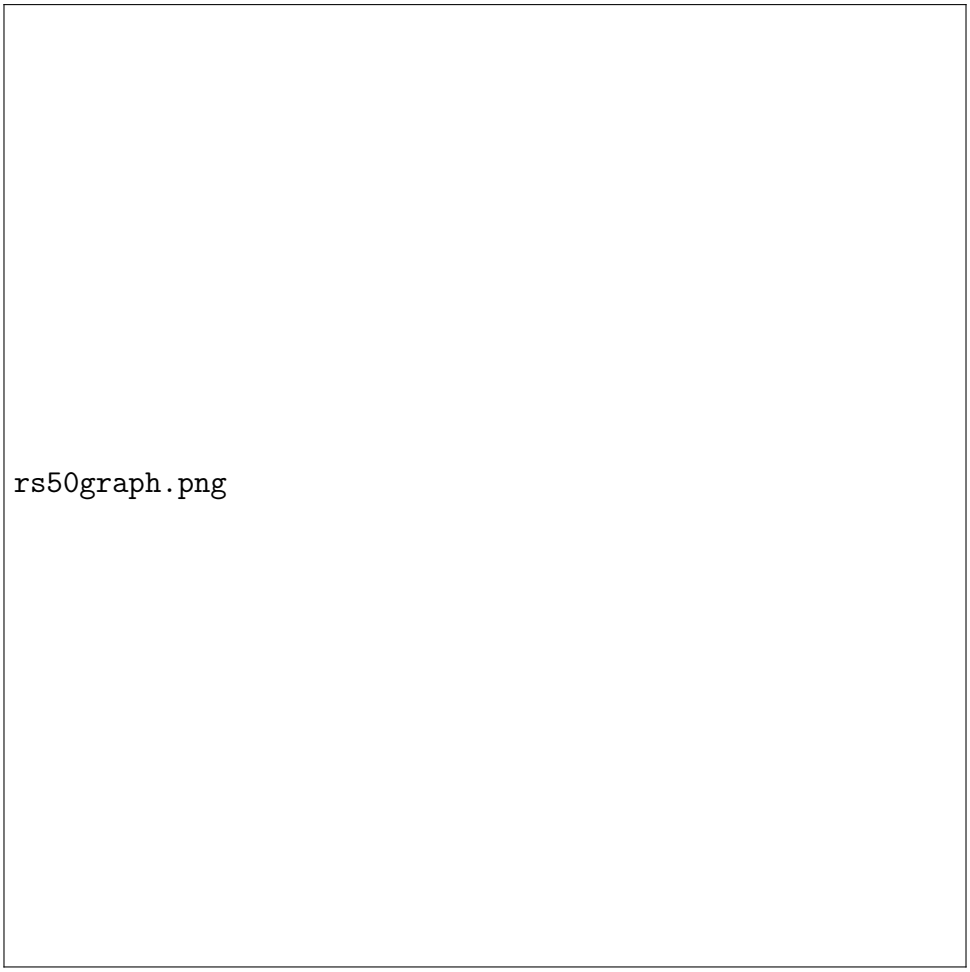
Model Summary:

Graphs:

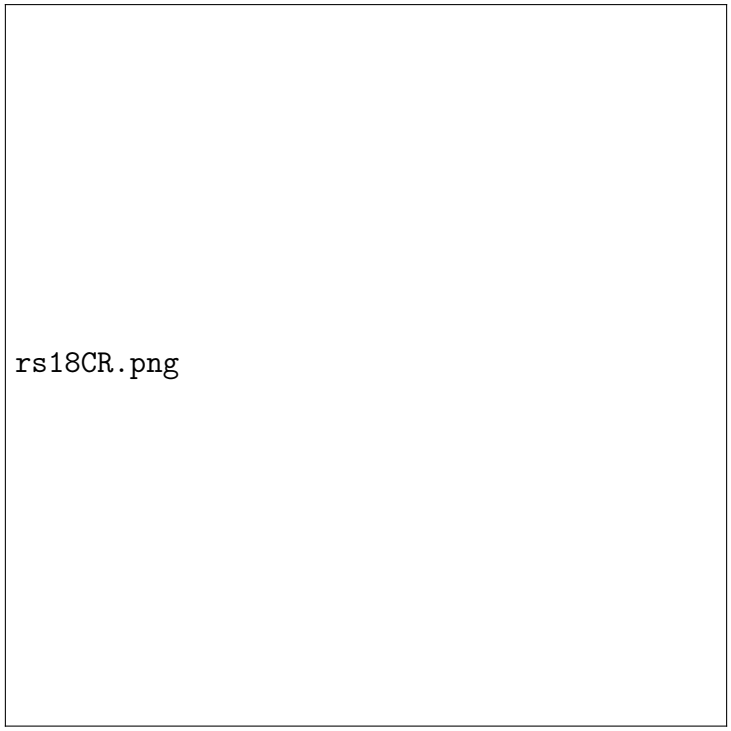
Accuracy and Loss Values:

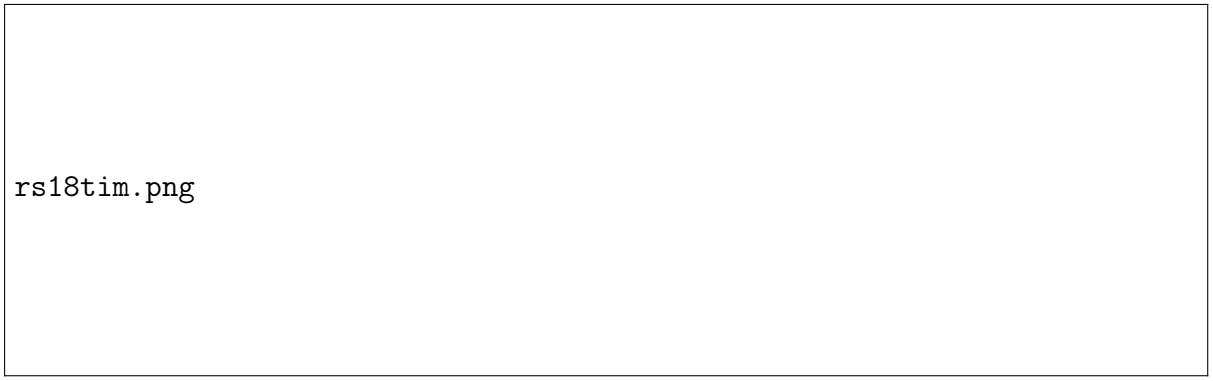



```
Final Training Accuracy: 51.95%
Final Training Loss: 1.1878
Final Validation Accuracy: 57.35%
Final Validation Loss: 1.1089
Test Accuracy: 54.67%
Test Loss: 1.1383
```



Confusion Matrix:





Classification Report:

Testing Images:

Architectural Comparision

Aspect	ResNet-18	ResNet-50
Number of Layers	18	50
Type of Block	Basic Block	Bottleneck Block
Parameters (Approx.)	11.2 million	25.6 million
Depth	Shallow	Deep
Inference Speed	Faster	Slower
Model Size	Smaller	Larger
Feature Learning	Basic Features	More Complex Features
Suitable For	Lightweight models	High-accuracy models

Performance Comparision

Metric	ResNet-18	ResNet-50
Training Accuracy	96.61%	98.20%
Validation Accuracy	94.24%	96.00%
Test Accuracy	96.93%	97.50%
Training Loss	0.6580	0.4205
Validation Loss	0.7089	0.5381
Test Loss	0.6793	0.5127
Parameters	11.2M	25.6M
Inference Time (per image)	Lower	Higher

Hyperparameters for ResNet Architectures

- **Input size:** $224 \times 224 \times 3$ (RGB images)
- **Optimizer:** SGD with momentum (momentum = 0.9) or Adam optimizer
- **Epochs:** Typically 50–100 (based on dataset and convergence)
- **Batch size:** Between 32 and 256 (based on GPU memory)
- **Activation function:** ReLU (Rectified Linear Unit)
- **Learning rate:** Initially set to 0.01 (adjusted with scheduler)
- **Loss function:** Categorical Cross-Entropy Loss
- **Dropout:** Typically 0.3–0.5 in fully connected layers

Conclusion:

- **ResNet18** is a shallower architecture that is faster to train and requires fewer resources. It is well-suited for smaller datasets or when computational efficiency is a priority.
- **ResNet50** introduces deeper layers with bottleneck blocks, allowing the model to learn more complex and hierarchical features. It generally provides better accuracy on large and complex datasets.
- If training speed and model simplicity are important, **ResNet18** is preferred.

- If higher accuracy and deeper representation are essential, especially for large-scale or fine-grained classification, **ResNet50** is the better choice.

.

11. RNN-Recurrent Neural Networks

Overview

- Why RNN needed for the new world
- hyperparameters we observed
- The difference we have from knn and k-means
- decision tree algorithm

11.1 Why changing from ANN to RNN for new

Recurrent Neural Networks (RNNs) were introduced to address the limitations of traditional neural networks, such as FeedForward Neural Networks (FNNs), when it comes to processing sequential data. FNN takes inputs and process each input independently through a number of hidden layers without considering the order and context of other inputs. Due to which it is unable to handle sequential data effectively and capture the dependencies between inputs. As a result, FNNs are not well-suited for sequential processing tasks such as, language modeling, machine translation, speech recognition, time series analysis, and many other applications that requires sequential processing. To address the limitations posed by traditional neural networks, RNN comes into the picture.

11.2 RNN Architecture

Recurrent Neural Networks (RNNs) are a type of neural network with hidden states and allow past outputs to be used as inputs. They usually go like this:

Here's a breakdown of its key components:

1. **Input Layer:** This layer receives the initial element of the sequence data. For example, in a sentence, it might receive the first word as a vector representation.

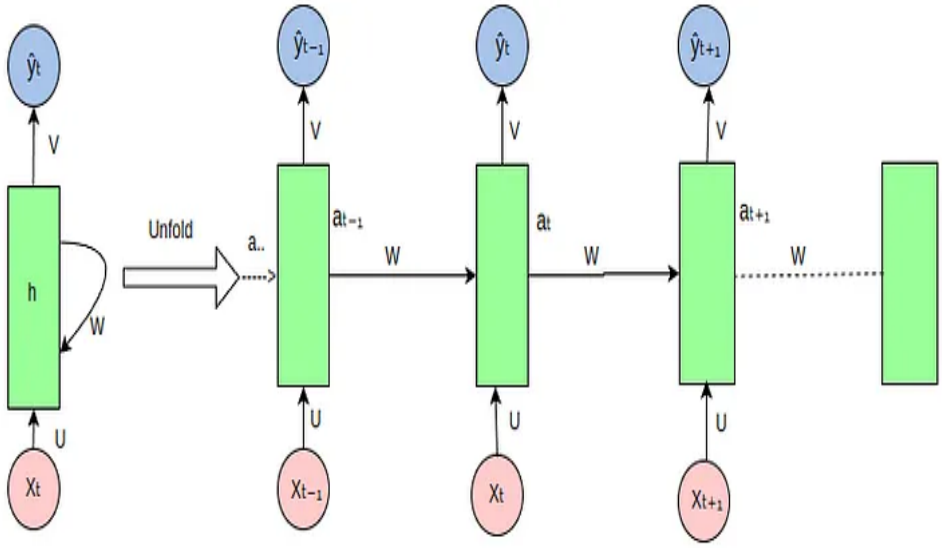


Figure 11.1: Classical v/s ML

2. **Hidden Layer:** The heart of the RNN, the hidden layer contains a set of interconnected neurons. Each neuron processes the current input along with the information from the previous hidden layer's state. This "state" captures the network's memory of past inputs, allowing it to understand the current element in context.

3. **Activation Function:** This function introduces non-linearity into the network, enabling it to learn complex patterns. It transforms the combined input from the current input layer and the previous hidden layer state before passing it on.

4. **Output Layer:** The output layer generates the network's prediction based on the processed information. In a language model, it might predict the next word in the sequence.

5. **Recurrent Connection:** A key distinction of RNNs is the recurrent connection within the hidden layer. This connection allows the network to pass the hidden state information (the network's memory) to the next time step. It's like passing a baton in a relay race, carrying information about previous inputs forward.

RNN overcome these limitations by introducing a recurrent connection that allow information to flow from one time-step to the next. This recurrent connection enables RNNs to maintain internal memory, where the output of each step is fed back as an input to the next step, allowing the network to capture the information from previous steps and utilize it in the current step, enabling model to learn temporal dependencies and handle input of variable length.

11.2.1 calculation of forward propagation at stages

The RNN takes an input vector X and the network generates an output vector y by scanning the data sequentially from left to right, with each time step updating the hidden state and producing an output. It shares the same parameters across all time steps. This means that, the same set of parameters, represented by U, V, W is used consistently throughout the network. U represents the weight parameter governing the connection from input layer X to the hidden layer h , W represents the weight associated with the connection between hidden layers, and V for the connection from hidden layer h to output layer y. This sharing of parameters allows the RNN to effectively capture temporal dependencies and process sequential data more efficiently by retaining the information from previous input in its current hidden state.

At each time step t, the hidden state a_t is computed based on the current input x_t , previous hidden state a_{t-1} and model parameters as illustrated by the following formula:

Equation (1)

$$a_t = f(a_{t-1}, x_t; \theta)$$

It can also be written as,

Alternate form of Equation (1)

$$a_t = f(U \cdot X_t + W \cdot a_{t-1} + b)$$

where,

a_t represents the output generated from the hidden layer at time step t.

x_t is the input at time step t.

θ represents a set of learnable parameters (weights and biases).

U is the weight matrix governing the connections from the input to the hidden layer; $U \in \theta$

W is the weight matrix governing the connections from the hidden layer to itself (recurrent connections); $W \in \theta$

V represents the weight associated with connection between hidden layer and output layer; $V \in \theta$

a_{t-1} is the output from hidden layer at time t-1.

b is the bias vector for the hidden layer; $b \in \theta$

f is the activation function.

For a finite number of time steps T=4, we can expand the computation graph of a Recurrent Neural Network, illustrated in Figure 3, by applying the equation (1) T-1

times.

Equation (2)

$$a_4 = f(a_3, x_4; \theta)$$

Equation (2) can be expanded as,

Expanded Computation for T

$$a_4 = f(U \cdot X_4 + W \cdot a_3 + b)$$
$$a_3 = f(U \cdot X_3 + W \cdot a_2 + b)$$
$$a_2 = f(U \cdot X_2 + W \cdot a_1 + b)$$

The output at each time step t, denoted as $\hat{y}_t$ is computed based on the hidden state output a_t using the following formula,

Equation (3)

$$\hat{y}_t = f(a_t; \theta)$$

Equation (3) can be written as,

Alternate form of Equation (3)

$$\hat{y}_t = f(V \cdot a_t + c)$$

when $t = 4$,

Prediction at t

$$\hat{y}_4 = f(V \cdot a_4 + c)$$

where,

$\hat{y}_t$ is the output predicted at time step t.

V is the weight matrix governing the connections from the hidden layer to the output layer.

c is the bias vector for the output layer.

11.3 Processing in RNN

Step-by-Step Processing in RNNs

- Input and Hidden Layer Dance: At each time step, the RNN receives an input element from the sequence. This input is combined with the information retained by the hidden layer from the previous step.
- Activation Function Magic: The combined data is processed by the hidden layer using an activation function. This function extracts meaningful patterns from the data.
- Feedback Loop for Memory: A crucial aspect of RNNs is the feedback loop within the hidden layer. This loop allows the network to store information from past time steps, enabling it to understand the context of the current input.
- Output Generation: Based on the processed information in the hidden layer, the RNN generates an output for the current time step. This output could be a prediction, a classification, or another element in the sequence.

11.4 Backpropagation Through Time (BPTT)

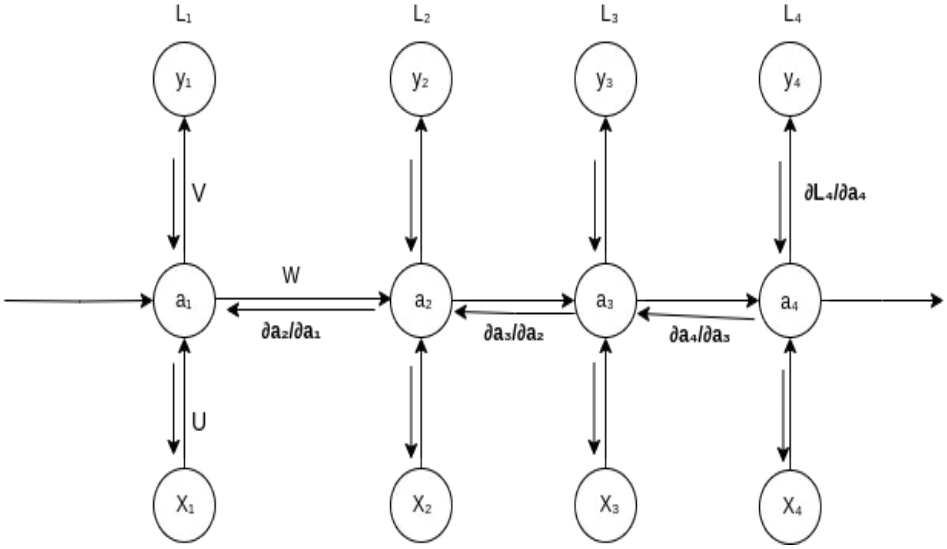


Figure 11.2: Classical v/s ML

The neural network in an RNN is organized, and since an ordered network computes variables one at a time in a predetermined order, such as first h_1, then h_2, then h_3, and so on, each variable is calculated individually. Therefore, we will use backpropagation in each of these concealed temporal stages in turn.

$L(\theta)$ (loss function) depends on h_3
h_3 in turn depends on h_2 and W
h_2 in turn depends on h_1 and W

h_1 in turn depends on h_0 and W
where h_0 is a constant starting state.
Therefore, the equation becomes:

$$\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial \theta}$$

Next, we simplify the equation by applying backpropagation on only one row

$$\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial \theta}$$

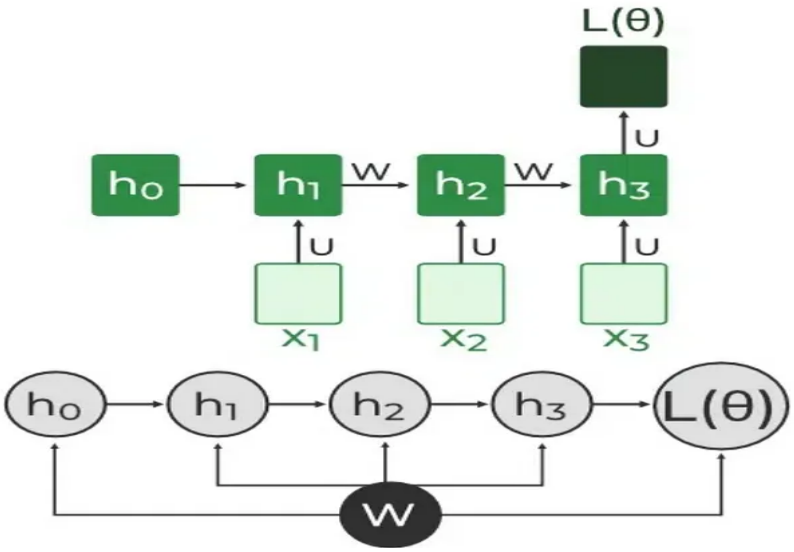


Figure 11.3: Classical v/s ML

Since this computation is the same as any other basic backpropagation in a deep neural network, we already know how to do it.

However, we will see how to apply backpropagation to this term

$$\frac{\partial h_3}{\partial \theta}$$

Since we know that,

$$h_3 = \sigma(W h_2 + b)$$

And in such an ordered network, we can't compute

$$\frac{\partial h_3}{\partial \theta}$$

directly

We will have to do it this way:

$$\frac{\partial h_3}{\partial \theta} = \frac{\partial h_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial \theta}$$

Let us simplify this further:

$$\frac{\partial h_2}{\partial \theta} = \frac{\partial h_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial \theta}$$

Finally, we get

$$\frac{\partial h_3}{\partial \theta} = \frac{\partial h_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial \theta}$$

This will be the endpoint

$$\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial \theta}$$

This is known as **backpropagation through time (BPTT)** as we backpropagate over all previous time steps.

11.5 Python example on RNN on stock price Prediction of TATA MOTORS in a span of year June2024-June2025

Listing 11.1: importing libraries

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
from tensorflow.keras.callbacks import EarlyStopping
```

Listing 11.2: explanation

```
1)pandas, numpy: For data handling and numerical computations.

2)matplotlib.pyplot: Used to visualize data and plot graphs.
```

```
3)MinMaxScaler: Scales data between 0 and 1 important for training neural networks.

4)mean_squared_error, mean_absolute_error, r2_score: Evaluation metrics to assess
    ↪ model accuracy.

5)tensorflow.keras: For building the RNN model.

6)Sequential: Container for stacking layers.

7)SimpleRNN: Basic RNN layer.

8)Dense: Fully connected layer for output.

9)EarlyStopping: Stops training if the model stops improving (prevents overfitting).
```

We have a dataset that has the stock prices of TATA MOTORS nearly one year data and we have done the prediction on that

Listing 11.3: importing libraries

```
file_path = '/content/drive/MyDrive/Quote-Equity-TATAMOTORS-EQ-06-06-2024-to
    ↪ -06-06-2025.csv'
df = pd.read_csv(file_path, thousands=',')

print(df.columns)
df.tail()
```

Index	Date	Series	Open	High	Low	Prev Close	LTP	Close	VWAP	52W
244	12-Jun-2024	EQ	994.5	1010.25	987.0	987.1	989.2	988.7	996.34	10
245	11-Jun-2024	EQ	973.8	992.55	966.65	975.15	986.15	987.1	984.93	10
246	10-Jun-2024	EQ	977.0	984.9	969.1	970.5	974.8	975.15	976.04	10
247	07-Jun-2024	EQ	940.0	973.0	935.25	938.25	970.0	970.5	959.91	10
248	06-Jun-2024	EQ	940.0	946.0	931.2	929.95	939.5	938.25	939.29	10

Table 11.1: Stock Price Data from 06-Jun-2024 to 12-Jun-2024

Listing 11.4: explanation

```
1)Reads the CSV file from Google Drive.
thousands=',': Handles numbers like 1,000 properly.

2)print(df.columns): Displays the column names in the dataset.
df.tail(): Shows the last few rows of the dataset to get a quick look at the data.
```

Listing 11.5: renaming

```
    df = df.rename(columns=lambda x: x.strip().lower())
df['date'] = pd.to_datetime(df['date'], format='%d-%b-%Y')
df = df.sort_values('date')
```

```
Cleans column names (removes spaces, converts to lowercase),

Converts the 'date' column to a datetime format,

Sorts the entire DataFrame by that date.
```

```
    df = df[['date', 'close']]
df = df.dropna()
```

Listing 11.6: explanation

```
1)Keeps only the 'date' and 'close' columns in the DataFrame.
To focus analysis only on these two relevant columns and discard all others.

2)Removes any rows that contain missing (NaN) values.
To ensure the data is clean and complete for analysis or modeling
```

Listing 11.7: renaming

```

    df = df.rename(columns=lambda x: x.strip().lower())
df['date'] = pd.to_datetime(df['date'], format='%d-%b-%Y')
df = df.sort_values('date')
```

```
Cleans column names (removes spaces, converts to lowercase),

Converts the 'date' column to a datetime format,

Sorts the entire DataFrame by that date.
```

Figure 11.4: Classical v/s ML

Listing 11.8: Normalisation

```
 scaler = MinMaxScaler()
df['close_scaled'] = scaler.fit_transform(df[['close']])
print("Before Normalization:\n", df[['close']].head())
print("After Normalization:\n", df[['close_scaled']].head())
```

```
This makes the values normalised as follows and prints the top data as in the sheet by
    ↪ the command dp.head()
Before Normalization:
      close
248 938.25
247 970.50
246 975.15
245 987.10
244 988.70
After Normalization:
     close_scaled
248 0.615874
247 0.671276
246 0.679265
245 0.699794
244 0.702543
```

Listing 11.9: renaming

```
 scaler = MinMaxScaler()
df['close_scaled'] = scaler.fit_transform(df[['close']])
print("Before Normalization:\n", df[['close']].head())
print("After Normalization:\n", df[['close_scaled']].head())
```

Listing 11.10: Explanation

```
 1)Prepares input sequences (X) and target values (y) for time series modeling (e.g.,
    ↪ LSTM).

2)X: sliding windows of past values (length = window_size)
y: the actual value that follows each window
data_scaled: the normalized 'close' prices

3)X, y: arrays ready for training (e.g., with LSTM)
Creates training sequences from the scaled closing prices:
Each X[i] is 10 past values
Each y[i] is the next value after that window
```

Listing 11.11: Data vector check

```
split = int(0.8 * len(X))
X_train, X_test = X[:split], X[split:]
```

```
y_train, y_test = y[:split], y[split:]
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
# y_test = y_test.re
```

Listing 11.12: Explanation

```
This gives the training and validation split and the .shape commands lets us print the
    ↪  vector value of the X_train.....we have
```

Listing 11.13: Model building

```
model = Sequential([
    SimpleRNN(239, activation='tanh', return_sequences=True, input_shape=(X_train.
        ↪ shape[1], 1)),
    SimpleRNN(239, activation='tanh', return_sequences=True),
    SimpleRNN(239, activation='tanh', return_sequences=True),
    # SimpleRNN(239, activation='tanh', return_sequences=True),
    # SimpleRNN(239, activation='tanh', return_sequences=True),
    SimpleRNN(239, activation='tanh', return_sequences=False),
    Dense(1)
])
  model.compile(optimizer='adam', loss='mean_squared_error',metrics=['accuracy'])
    model.summary()
```

Layer (type)	**Output Shape**	**Param #**
LSTM (lstm)	(None, 10, 239)	230,396
LSTM (lstm_1)	(None, 10, 239)	457,924
LSTM (lstm_2)	(None, 10, 239)	457,924
LSTM (lstm_3)	(None, 239)	457,924
Dense (dense)	(None, 1)	240
Total params		**1,604,408 (6.12 MB)**
Trainable params		**1,604,408 (6.12 MB)**
Non-trainable params		**0 (0.00 B)**

Table 11.2: Model Summary of the Sequential LSTM Network

Listing 11.14: Explanation

```
1)This model stacks 4 LSTM layers (each with 239 units) to capture time-based patterns
    ↪ , then ends with a Dense layer to predict the next value in the sequence.
2)return_sequences=True: outputs the full sequence to the next LSTM layer

3)when return_sequences=False Outputs only the last time step
```

Listing 11.15: Training

```
history = model.fit(
    X_train, y_train,
    validation_data=(X_test, y_test),
    epochs=100,
    # callbacks=[early_stop],
    verbose=1
)
```

Listing 11.16: Explanation

```
This part of code is the reason that led to the phase of training and getting the
    ↪ accuracy plot and we also used "early stop" that will see if the algorithm's
    ↪ loss is deacresing else it tolerates till a fixed numnber of epochs then it
    ↪ terminates
```

Listing 11.17: Accuracy Plot

```
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.title("Training vs Validation Loss")
plt.legend()
plt.show()
```

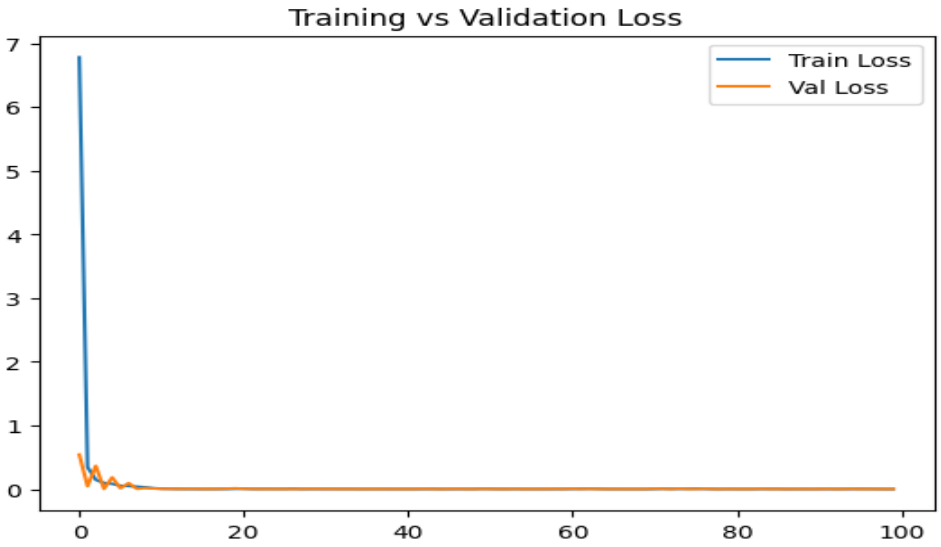


Figure 11.5: Classical v/s ML

Listing 11.18: Metrics

```
mae = mean_absolute_error(y_test_inv, y_pred_inv)
rmse = np.sqrt(mean_squared_error(y_test_inv, y_pred_inv))
r2 = r2_score(y_test_inv, y_pred_inv)
print(f"\nMAE: {mae:.2f}, RMSE: {rmse:.2f}, R_square Score: {r2:.2f}")
```

Listing 11.19: Output

```
MAE: 24.11, RMSE: 32.97, R_square Score: 0.43
```

Listing 11.20: Actual V/S Predicted

```
plt.figure(figsize=(10, 5))
plt.plot(y_test_inv, label='Actual')
plt.plot(y_pred_inv, label='Predicted')
plt.title("Predicted vs Actual Closing Prices")
plt.legend()
plt.grid(True)
plt.show()
```

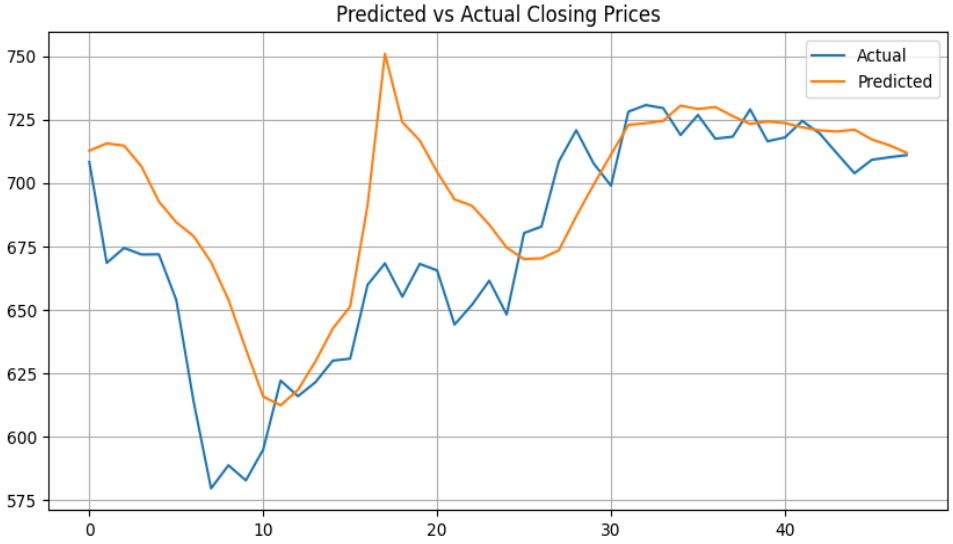


Figure 11.6: Classical v/s ML

This is all how we can build a RNN model almost everything remains same for the RNN and only the thing that changes can be the model code as of the architecture changed as of we can point it out

11.6 Why are we moving to LSTM

11.6.1 Recurrent Neural Networks (RNNs): Overview

Recurrent Neural Networks (RNNs) are widely used for sequential data such as text, speech, and time series. They are designed to carry information across time steps via hidden states. However, RNNs struggle with capturing long-term dependencies due to limitations like vanishing and exploding gradients.

11.6.2 The Short-Term Memory Problem

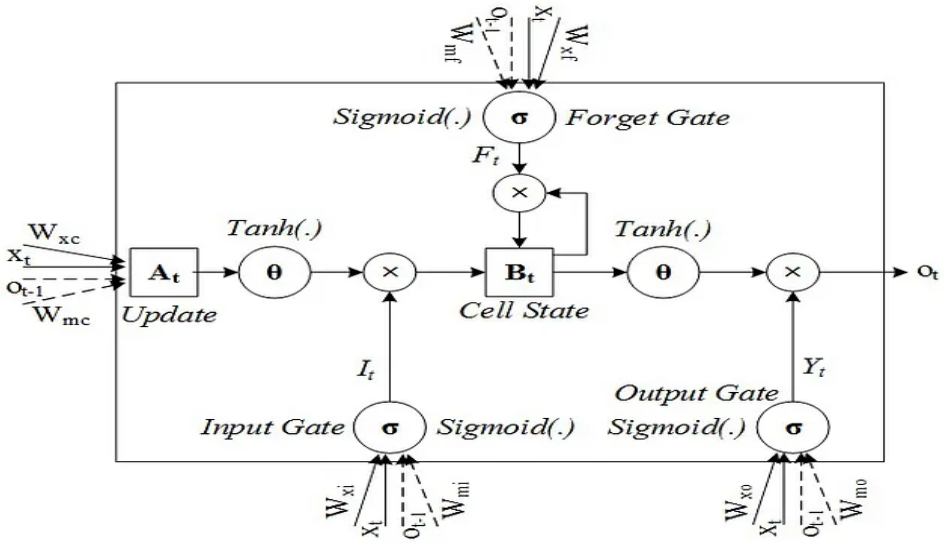


Figure 11.7: Classical v/s ML

RNNs inherently suffer from short-term memory. As the sequence length increases, it becomes difficult for the network to retain early time step information. For example, when processing a paragraph of text, RNNs may fail to capture critical context from the beginning.

11.6.3 Vanishing and Exploding Gradients

During training, an **error gradient** is used to update network weights. However, in deep or recurrent architectures, these gradients can either explode or vanish:

11.6.4 Exploding Gradients

When gradients grow excessively large during backpropagation, they cause large weight updates, destabilizing the network or resulting in `NaN` values. This typically happens when gradient values larger than 1.0 are repeatedly multiplied across layers.

11.6.5 Vanishing Gradients

Conversely, when gradients shrink as they propagate backward, earlier layers receive almost no update. This results in the network failing to learn long-range patterns. This usually occurs when gradient values less than 1.0 diminish exponentially across layers.

11.7 Detecting Gradient Issues

11.7.1 Signs of Exploding Gradients

- Training loss does not decrease effectively.
- Sudden spikes or instability in the loss curve.
- Loss becomes `NaN` during training.

11.7.2 Signs of Vanishing Gradients

- Kernel weights stop changing or shrink towards zero.
- Model shows poor learning, especially on earlier layers.
- Could be caused by poor data, choice of loss function, or optimizer.

Monitoring gradients using visualizations or checking gradient norms helps identify such issues.

11.8 Why LSTM Helps

Long Short-Term Memory (LSTM) networks were developed to overcome the gradient issues in standard RNNs. Using gating mechanisms (input, forget, and output gates), LSTMs preserve and regulate the flow of information across longer sequences, enabling effective learning of long-term dependencies.

11.9 Long Term Short Memory Architecture

11.9.1 LSTM Cell Components

An LSTM cell comprises three gates:

- **Forget Gate (F_t)**: Decides what information to discard from the cell state.
- **Input Gate (I_t)**: Determines which new information to add to the cell state.
- **Output Gate (O_t)**: Decides what information to output based on the cell state.

The cell state (C_t) acts as a conveyor belt, allowing information to flow unchanged unless explicitly modified by the gates. This structure enables gradients to pass through many time steps without vanishing or exploding, thus preserving long-term dependencies.

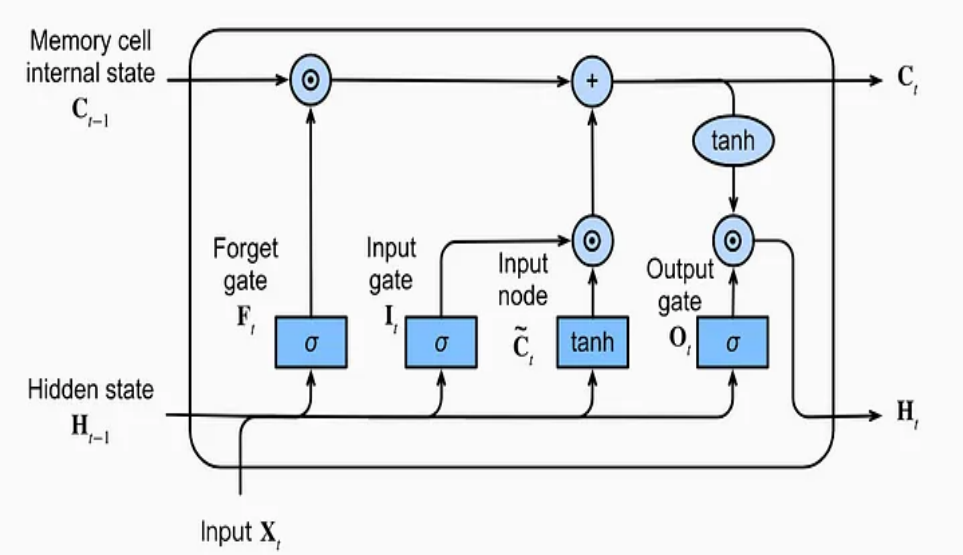


Figure 11.8: Classical v/s ML

Source: *10.1. Long Short-Term Memory (LSTM) — Dive into Deep Learning 1.0.3 documentation*

11.9.2 Information Flow in LSTM

1. **Receiving Inputs:** The LSTM unit receives x_t, h_{t-1}, and C_{t-1}.
2. **Forget Gate Processing:** Decides which information to retain in C_{t-1}.
3. **Input Gate and Candidate Memory:** Generates new candidate information $\tilde{C}_t$ and determines how much of it to add to the cell state.
4. **Updating Cell State:** Combines the old cell state and new candidate information to form C_t.
5. **Output Gate Processing:** Determines the new hidden state h_t based on the updated cell state.

11.10 Python example on LSTM on stock price Prediction of TATA MOTORS in a span of year June2024-June2025

Listing 11.21: importing libraries

```
import numpy as np
import matplotlib.pyplot as plt
```

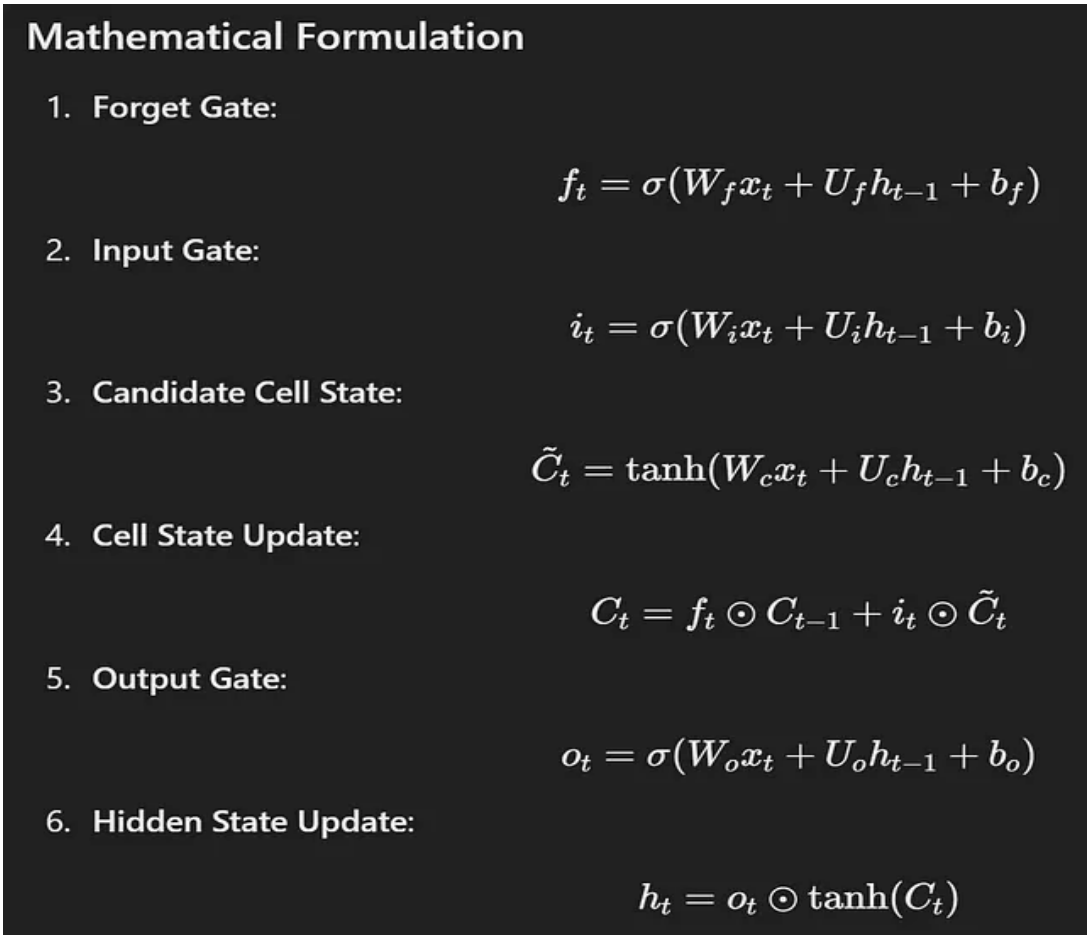


Figure 11.9: Classical v/s ML

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
from tensorflow.keras.callbacks import EarlyStopping
```

Listing 11.22: explanation

```
1)pandas, numpy: For data handling and numerical computations.

2)matplotlib.pyplot: Used to visualize data and plot graphs.
3)MinMaxScaler: Scales data between 0 and 1 important for training neural networks.

4)mean_squared_error, mean_absolute_error, r2_score: Evaluation metrics to assess
    ↪ model accuracy.

5)tensorflow.keras: For building the RNN model.

6)Sequential: Container for stacking layers.

7)SimpleRNN: Basic RNN layer.
```

```
8)Dense: Fully connected layer for output.

9)EarlyStopping: Stops training if the model stops improving (prevents overfitting).
```

We have a dataset that has the stock prices of TATA MOTORS nearly one year data and we have done the prediction on that

Listing 11.23: dataset

```
file_path = '/content/drive/MyDrive/Quote-Equity-TATAMOTORS-EQ-06-06-2024-to
    ↪ -06-06-2025.csv'
df = pd.read_csv(file_path, thousands=',')

print(df.columns)
df.tail()
```

Index	Date	Series	Open	High	Low	Prev Close	LTP	Close	VWAP	52W
244	12-Jun-2024	EQ	994.5	1010.25	987.0	987.1	989.2	988.7	996.34	10
245	11-Jun-2024	EQ	973.8	992.55	966.65	975.15	986.15	987.1	984.93	10
246	10-Jun-2024	EQ	977.0	984.9	969.1	970.5	974.8	975.15	976.04	10
247	07-Jun-2024	EQ	940.0	973.0	935.25	938.25	970.0	970.5	959.91	10
248	06-Jun-2024	EQ	940.0	946.0	931.2	929.95	939.5	938.25	939.29	10

Table 11.3: Stock Price Data from 06-Jun-2024 to 12-Jun-2024

Listing 11.24: explanation

```
1)Reads the CSV file from Google Drive.
thousands=',': Handles numbers like 1,000 properly.

2)print(df.columns): Displays the column names in the dataset.
df.tail(): Shows the last few rows of the dataset to get a quick look at the data.
```

Listing 11.25: renaming

```
    df = df.rename(columns=lambda x: x.strip().lower())
df['date'] = pd.to_datetime(df['date'], format='%d-%b-%Y')
df = df.sort_values('date')
```

```
Cleans column names (removes spaces, converts to lowercase),

Converts the 'date' column to a datetime format,

Sorts the entire DataFrame by that date.
```

```
    df = df[['date', 'close']]
df = df.dropna()
```

Listing 11.26: Explanation

```
1)Keeps only the 'date' and 'close' columns in the DataFrame.
To focus analysis only on these two relevant columns and discard all others.

2)Removes any rows that contain missing (NaN) values.
To ensure the data is clean and complete for analysis or modeling
```

Listing 11.27: renaming

```
    \begin{lstlisting}[caption=renaming]
    df = df.rename(columns=lambda x: x.strip().lower())
df['date'] = pd.to_datetime(df['date'], format='%d-%b-%Y')
df = df.sort_values('date')
```

Listing 11.28: Explanation

```
Cleans column names (removes spaces, converts to lowercase),

Converts the 'date' column to a datetime format,

Sorts the entire DataFrame by that date.
```

Figure 11.10: Classical v/s ML

Listing 11.29: Normalisation

```
 scaler = MinMaxScaler()
df['close_scaled'] = scaler.fit_transform(df[['close']])
print("Before Normalization:\n", df[['close']].head())
print("After Normalization:\n", df[['close_scaled']].head())
```

Listing 11.30: Explanation

```
This makes the values normalised as follows and prints the top data as in the sheet by
    ↪ the command dp.head()
Before Normalization:
      close
248 938.25
247 970.50
246 975.15
245 987.10
244 988.70
After Normalization:
     close_scaled
248 0.615874
247 0.671276
246 0.679265
245 0.699794
244 0.702543
```

Listing 11.31: renaming

```
def create_sequences(data, window_size):
    X, y = [], []
    for i in range(window_size, len(data)):
        X.append(data[i - window_size:i])
        y.append(data[i])
    return np.array(X), np.array(y)

window_size = 10
data_scaled = df['close_scaled'].values
X, y = create_sequences(data_scaled, window_size)
```

Listing 11.32: Explanation

```
 1)Prepares input sequences (X) and target values (y) for time series modeling (e.g.,
    ↪ LSTM).

2)X: sliding windows of past values (length = window_size)
y: the actual value that follows each window
data_scaled: the normalized 'close' prices

3)X, y: arrays ready for training (e.g., with LSTM)
Creates training sequences from the scaled closing prices:
Each X[i] is 10 past values
Each y[i] is the next value after that window
```

Listing 11.33: Checking dataset vectors

```
split = int(0.8 * len(X))
```

```
X_train, X_test = X[:split], X[split:]
y_train, y_test = y[:split], y[split:]
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

Listing 11.34: Explanation

```
This gives the training and validation split and the .shape commands lets us print the
    ↪ vector value of the X_train.....we have
```

Listing 11.35: Model building

```
model = Sequential([
    LSTM(239, activation='tanh', return_sequences=True, input_shape=(X_train.shape[1],
        ↪ 1)),
    LSTM(239, activation='tanh', return_sequences=True),
    LSTM(239, activation='tanh', return_sequences=True),
    # LSTM(239, activation='tanh', return_sequences=True),
    # LSTM(239, activation='tanh', return_sequences=True),
    # LSTM(239, activation='tanh', return_sequences=True),
    LSTM(239, activation='tanh', return_sequences=False),
    # Dense(239, activation='tanh'),
    Dense(1)
])
  model.compile(optimizer='adam', loss='mean_squared_error',metrics=['accuracy'])
    model.summary()
```

Layer (type)	**Output Shape**	**Param #**
SimpleRNN (simple_rnn_6)	(None, 10, 239)	57,599
SimpleRNN (simple_rnn_7)	(None, 10, 239)	114,481
SimpleRNN (simple_rnn_8)	(None, 10, 239)	114,481
SimpleRNN (simple_rnn_9)	(None, 239)	114,481
Dense (dense_1)	(None, 1)	240
Total params		**401,282 (1.53 MB)**
Trainable params		**401,282 (1.53 MB)**
Non-trainable params		**0 (0.00 B)**

Table 11.4: Model Summary of the Sequential SimpleRNN Network

Listing 11.36: Explanation

```
1)This model stacks 4 LSTM layers (each with 239 units) to capture time-based patterns
    ↪ , then ends with a Dense layer to predict the next value in the sequence.
2)return_sequences=True: outputs the full sequence to the next LSTM layer

3)when return_sequences=False Outputs only the last time step
```

Listing 11.37: Training

```
history = model.fit(
    X_train, y_train,
    validation_data=(X_test, y_test),
    epochs=100,
    # callbacks=[early_stop],
    verbose=1
)
```

Listing 11.38: Explanation

```
This part of code is the reason that led to the phase of training and getting the
    ↪ accuracy plot and we also used "early stop" that will see if the algorithm's
    ↪ loss is deacresing else it tolerates till a fixed numnber of epochs then it
    ↪ terminates
```

Listing 11.39: Accuracy plotting

```
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.title("Training vs Validation Loss")
plt.legend()
plt.show()
```

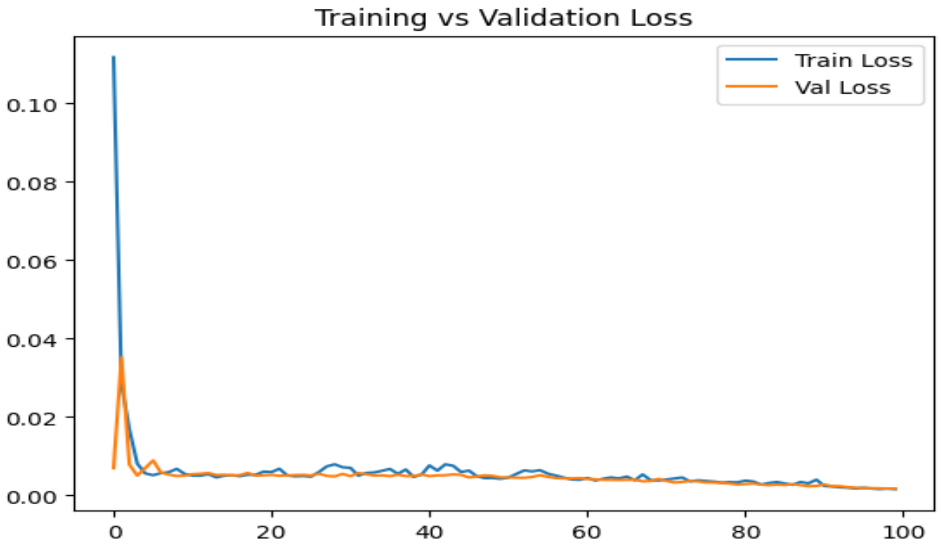


Figure 11.11: Classical v/s ML

Listing 11.40: Metrics

```
mae = mean_absolute_error(y_test_inv, y_pred_inv)
rmse = np.sqrt(mean_squared_error(y_test_inv, y_pred_inv))
r2 = r2_score(y_test_inv, y_pred_inv)
print(f"\n MAE: {mae:.2f}, RMSE: {rmse:.2f}, R_square Score: {r2:.2f}")
```

Listing 11.41: Explanation

```
MAE: 17.43, RMSE: 23.97, R_square Score: 0.70
```

Listing 11.42: Actual V/S Real

```
plt.figure(figsize=(10, 5))
plt.plot(y_test_inv, label='Actual')
plt.plot(y_pred_inv, label='Predicted')
plt.title("Predicted vs Actual Closing Prices")
plt.legend()
plt.grid(True)
plt.show()
```

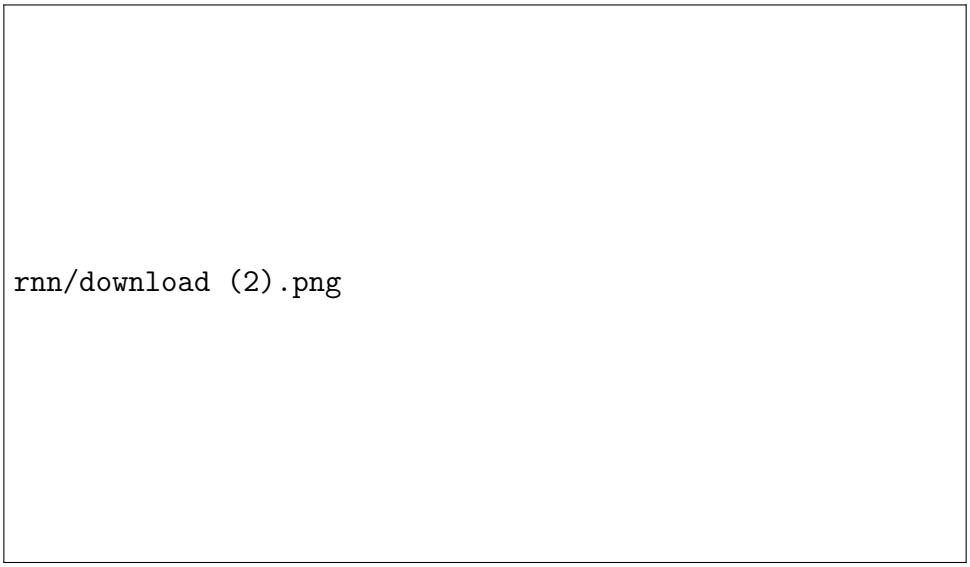


Figure 11.12: Classical v/s ML

This is all how we can build a LSTM model almost everything remains same for the RNN and only the thing that changes can be the model code as of the architecture changed as of we can point it out

11.11 Conclusion

Clearly we can see the difference in the R2 value of the two models made for the same dataset and this is what lets us developing in todays generation as the need of no-linearity with the the reqquirement to remember the previous data is also shown in this Stock prediction data