Experiment-1

Aim: Setting up the anaconda environment and executing python program.

Theory:

1. Introduction to Anaconda

Anaconda is a widely used open-source distribution platform for **Python** and **R** programming languages, primarily designed for **data science**, **machine learning**, and **scientific computing**. It simplifies the process of managing dependencies, packages, and environments, allowing users to isolate and control different project setups.
Anaconda includes several key tools such as:

- **Conda** – a package, dependency, and environment manager.
- **Anaconda Navigator** – a graphical interface to launch applications and manage environments.
- **Spyder**, **Jupyter Notebook**, and **VS Code** integrations for efficient code development and execution.

Using Anaconda ensures that project libraries and Python versions do not conflict, which is particularly useful when working across multiple projects or research tasks.

2. Python Programming Environment

Python is a high-level, interpreted language known for its simplicity and readability. When executed through Anaconda, Python programs run within virtual environments that maintain specific package versions and dependencies.
This controlled environment enhances reproducibility and reliability, two critical aspects of scientific and academic computing.

Code & Output :

```
C:\Users\Admin\OneDrive\Desktop\my dsktop\college\sem7\rldl>conda --version
conda 24.5.0

C:\Users\Admin\OneDrive\Desktop\my dsktop\college\sem7\rldl>conda create -n myenv python=3.10
Channels:
 - defaults
Platform: win-64
Collecting package metadata (repodata.json): done
Solving environment: done

Downloading and Extracting Packages:

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate myenv
#
# To deactivate an active environment, use
#
#     $ conda deactivate


C:\Users\Admin\OneDrive\Desktop\my dsktop\college\sem7\rldl>conda activate mymyenv
```

```
>>> def fibo(n):
...     seq = []
...     a,b = 0,1
...     for _ in range(n):
...             seq.append(a)
...             a,b = b, a+b
...     return seq
...
>>> print(fibo(10))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
>>> exit()

(myenv) C:\Users\Admin\OneDrive\Desktop\my dsktop\college\sem7\rldl>conda deactivate

C:\Users\Admin\OneDrive\Desktop\my dsktop\college\sem7\rldl>
```

Learning outcome:

Experiment – 2

Aim: Installing Keras, Tensorflow and Pytorch libraries and making use of them.

Theory:

1. Introduction

In modern machine learning and deep learning workflows, specialized Python libraries such as **Keras**, **TensorFlow**, and **PyTorch** play a pivotal role in building, training, and deploying neural network models.
This experiment focuses on the **installation** and **basic utilization** of these frameworks within a controlled environment (e.g., Anaconda), which simplifies package management and ensures compatibility between dependencies.

2. Overview of Deep Learning Frameworks

(a) TensorFlow

TensorFlow, developed by **Google Brain**, is an open-source deep learning framework designed for numerical computation and large-scale machine learning.
It provides a flexible architecture that allows for easy deployment on CPUs, GPUs, and TPUs.
TensorFlow's core concepts include:

- **Tensors:** Multidimensional data arrays (similar to NumPy arrays).
- **Computation Graphs:** Mathematical models representing operations.
- **Automatic Differentiation:** Enables backpropagation for training neural networks.
- **Keras API:** A high-level interface integrated into TensorFlow for building neural networks more easily.

(b) Keras

Keras is a **high-level neural network API** that provides a user-friendly interface for defining, training, and evaluating deep learning models.
Initially developed as an independent project, it is now fully integrated into TensorFlow (tensorflow.keras).
Key features of Keras include:

- Simplicity and modularity.
- Support for both **Sequential** and **Functional** APIs.
- Built-in support for common layers, optimizers, and loss functions.
- Easy model visualization and debugging.

(c) PyTorch

PyTorch, developed by **Faceb**ook's AI Research Lab (FAIR), is another powerful open-source deep learning library. It is known for its dynamic computation graph, which allows on-the-fly modifications of model architectures — making it extremely flexible for research.
Key features include:

- **Tensors and Autograd:** Core components that enable gradient-based optimization.
- **Dynamic Graphs:** Easier debugging and model experimentation.
- **GPU Acceleration:** Efficient execution on CUDA-enabled devices.
- **Torchvision:** A module for common datasets and image transformations.

Code and Output:

```
import tensorflow as tf
import torch
print("TensorFlow version:", tf.__version__)
print("Keras version:", tf.keras.__version__)
print("PyTorch version:", torch.__version__)
print("CUDA available:", torch.cuda.is_available())
```

```
TensorFlow version: 2.19.0
Keras version: 3.10.0
PyTorch version: 2.8.0+cu126
CUDA available: False
```

```
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np

# Build a simple neural network
model = keras.Sequential([
    layers.Dense(64, activation='relu', input_shape=(100,)),
    layers.Dense(10, activation='softmax')
])
```

```
# Compile
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

print(model.summary())
```

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
dense_2 (Dense)	(None, 64)	6,464
dense_3 (Dense)	(None, 10)	650

```
Total params: 7,114 (27.79 KB)
Trainable params: 7,114 (27.79 KB)
Non-trainable params: 0 (0.00 B)
```

```python
import torch
import torch.nn as nn
import torch.optim as optim

# Dummy dataset
X_train = torch.rand(1000, 100)   # 1000 samples, 100 features
y_train = torch.randint(0, 10, (1000,))  # 10 classes

# Define simple network
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(100, 64)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(64, 10)

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = SimpleNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```python
# Training loop
for epoch in range(5):  # 5 epochs
    optimizer.zero_grad()
    outputs = model(X_train)
    loss = criterion(outputs, y_train)
    loss.backward()
    optimizer.step()
    print(f"Epoch {epoch+1}, Loss: {loss.item():.4f}")

# Evaluation
with torch.no_grad():
    outputs = model(X_train)
    _, predicted = torch.max(outputs, 1)
    accuracy = (predicted == y_train).float().mean()
    print(f"Training Accuracy: {accuracy:.4f}")
```

```
Epoch 1, Loss: 2.3099
Epoch 2, Loss: 2.3048
Epoch 3, Loss: 2.3010
Epoch 4, Loss: 2.2980
Epoch 5, Loss: 2.2957
Training Accuracy: 0.1180
```

Learning outcome:

Experiment – 3

Aim: Implement Q-learning with pure Python to play a game

Theory:

1. Introduction

Q-learning is a foundational algorithm in **Reinforcement Learning (RL)** — a branch of machine learning concerned with how agents take actions in an environment to maximize cumulative rewards.
Unlike supervised learning, where models learn from labeled data, reinforcement learning involves an **agent** learning by **interacting** with its environment through **trial and error**.
This experiment aims to implement the **Q-learning algorithm using pure Python**, without relying on external libraries such as TensorFlow or PyTorch, and to use it to play a simple game (e.g., grid navigation or maze solving).

2. Theoretical Background

(a) Reinforcement Learning Basics

Reinforcement learning consists of four key components:

1. **Agent:** The learner or decision-maker.
2. **Environment:** The world in which the agent operates.
3. **State (S):** A representation of the environment at a specific time.
4. **Action (A):** The set of possible moves the agent can make.
5. **Reward (R):** Feedback signal received after taking an action.

The agent learns a **policy** — a mapping from states to actions — that maximizes long-term reward.

(b) Q-Learning Algorithm

Q-learning is a type of **model-free reinforcement learning algorithm** that uses a **Q-table** to learn the value (expected future reward) of taking a particular action in a given state.
The **Q-value update rule** is given by:

$$Q(s,a) = Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

Where:

- s: Current state
- a: Action taken
- r: Immediate reward received
- s′: Next state

- α: Learning rate (controls how much new information overrides old)
- γ: Discount factor (determines importance of future rewards)

The goal is to learn an **optimal Q-function**, $Q_*(s,a)$Q^*(s, a)$Q_*(s,a)$, that gives the maximum expected reward for each state-action pair.

3. Working Principle

1. **Initialize** the Q-table (usually with zeros).
2. For each episode:
    - Start from an initial state.
    - Choose an action using an **exploration strategy** (e.g., ε-greedy).
    - Perform the action and observe the new state and reward.
    - Update the Q-table using the Q-learning formula.
    - Repeat until reaching a terminal state (game over or goal).
3. After many episodes, the agent learns which actions yield the highest long-term rewards.

4. Implementation in Python

Using **pure Python**, the Q-learning algorithm can be implemented using:

- **Lists or dictionaries** to represent the Q-table.
- Simple **loops** for episodes and state transitions.
- An **ε-greedy policy** to balance exploration and exploitation.

Code :

```
!pip install gym numpy
!pip install gymnasium[all] numpy
import gymnasium as gym
import numpy as np
import time

# Create environment
env = gym.make("FrozenLake-v1", is_slippery=False)

q_table = np.zeros((env.observation_space.n, env.action_space.n))

# Hyperparameters
alpha = 0.8
gamma = 0.95
epsilon = 1.0
epsilon_decay = 0.995
epsilon_min = 0.01
episodes = 2000
max_steps = 100

# Training
for episode in range(episodes):
    state, info = env.reset()
    done = False
```

```
    for step in range(max_steps):
        if np.random.rand() < epsilon:
            action = env.action_space.sample()
        else:
            action = np.argmax(q_table[state])

        next_state, reward, terminated, truncated, info = env.step(action)
        done = terminated or truncated

        # Q-learning update
        q_table[state, action] += alpha * (reward + gamma * np.max(q_table[next_state]) -
q_table[state, action])

        state = next_state
        if done:
            break

    epsilon = max(epsilon_min, epsilon * epsilon_decay)

print("Training finished.\n")

# Test
state, info = env.reset()
done = False
for step in range(max_steps):
    action = np.argmax(q_table[state])
    next_state, reward, terminated, truncated, info = env.step(action)
    done = terminated or truncated
    env.render()
    time.sleep(0.5)
    state = next_state
    if done:
        if reward > 0:
            print("Agent reached the goal!")
        else:
            print("Agent fell into a hole!")
        break

env.close()
```

Output:

```
Training finished.

/usr/local/lib/python3.12/dist-packages/gymnasiu
  gym.logger.warn(
/usr/local/lib/python3.12/dist-packages/jupyter_
  return datetime.utcnow().replace(tzinfo=utc)
Agent reached the goal!
```

Learning Outcome:

Experiment – 4

Aim: Python implementation of the iterative policy evaluation and update

Theory:

1. Introduction

In **Reinforcement Learning (RL)**, one of the key objectives is to determine an **optimal policy** that maximizes the expected cumulative reward for an agent interacting with an environment.

Iterative policy evaluation and update is a **Dynamic Programming (DP)** approach used to:

1. **Evaluate a given policy** — calculating the value function V(s)V(s)V(s) for all states.
2. **Improve the policy** — updating it based on the value function to move toward an optimal policy.

This experiment focuses on implementing these steps using **pure Python**, enabling students to understand the mechanics behind policy evaluation and improvement without relying on advanced libraries.

---

2. Theoretical Background

(a) Policy

A **policy** π is a mapping from states $s \in S$ to actions $a \in A$:

$$\pi(s) = a$$

- Deterministic Policy: Single action per state.
- Stochastic Policy: Probabilistic mapping of actions to states.

(b) Value Function

The **state-value function** $V\pi(s)$ represents the expected cumulative reward starting from state s and following policy π:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^{t} R_{t+1} \mid S_0 = s\right]$$

Where:

- γ is the discount factor ($0 \leq \gamma \leq 1$).
- R{t+1} is the reward at time step t+1.

(c) Iterative Policy Evaluation

This method **iteratively updates the value function** for all states using the Bellman expectation equation until convergence:

$$V_{k+1}(s) = \sum_{a} \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma V_k(s')\right]$$

Where:

- $p(s',r|\ s,a)$ is the probability of transitioning to state s' with reward r after taking action a in state s.

(d) Policy Improvement

Once the value function $V\pi(s)$ is estimated, a new **improved policy** is derived by choosing actions that maximize expected returns:

$$\pi_{\text{new}}(s) = \arg\max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma V(s')\right]$$

Repeating **evaluation** and **improvement** iteratively eventually converges to the **optimal policy** π^*.

Code :

```
import numpy as np

# Define Gridworld environment
class Gridworld:
    def __init__(self, n=4, gamma=1.0):
        self.n = n
        self.gamma = gamma
        self.states = [(i, j) for i in range(n) for j in range(n)]
        self.actions = ['U', 'D', 'L', 'R']
        self.terminal_states = [(0, 0), (n - 1, n - 1)]

    def step(self, state, action):
        if state in self.terminal_states:
            return state, 0

        i, j = state
        if action == 'U':
            i = max(i - 1, 0)
        elif action == 'D':
            i = min(i + 1, self.n - 1)
        elif action == 'L':
            j = max(j - 1, 0)
        elif action == 'R':
            j = min(j + 1, self.n - 1)

        next_state = (i, j)
        reward = -1  # small penalty to encourage shorter paths
        return next_state, reward

# Policy evaluation
def policy_evaluation(env, policy, theta=1e-6, gamma=1.0):
    V = {s: 0 for s in env.states}  # initialize value function

    while True:
        delta = 0
        for state in env.states:
            if state in env.terminal_states:
                continue

            v = V[state]
            new_v = 0
            for action, action_prob in policy[state].items():
```

```
                next_state, reward = env.step(state, action)
                new_v += action_prob * (reward + gamma * V[next_state])

            V[state] = new_v
            delta = max(delta, abs(v - new_v))

        if delta < theta:
            break
    return V

# Policy update (greedy improvement)
def policy_improvement(env, V, gamma=1.0):
    policy = {}
    for state in env.states:
        if state in env.terminal_states:
            policy[state] = {}
            continue

        q_values = {}
        for action in env.actions:
            next_state, reward = env.step(state, action)
            q_values[action] = reward + gamma * V[next_state]

        # greedy action(s)
        max_q = max(q_values.values())
        best_actions = [a for a, q in q_values.items() if q == max_q]

        # assign equal prob to best actions
        policy[state] = {a: 1/len(best_actions) for a in best_actions}
    return policy

# Policy Iteration (evaluation + improvement loop)
def policy_iteration(env, gamma=1.0, theta=1e-6):
    # start with uniform random policy
    policy = {s: {a: 1/len(env.actions) for a in env.actions} for s in env.states}

    while True:
        V = policy_evaluation(env, policy, theta, gamma)
        new_policy = policy_improvement(env, V, gamma)

        if new_policy == policy:
            break
        policy = new_policy

    return policy, V

# ---------------------- RUN ----------------------
env = Gridworld(n=4, gamma=1.0)
optimal_policy, optimal_value = policy_iteration(env)

print("Optimal Value Function:")
for i in range(env.n):
    print([round(optimal_value[(i, j)], 2) for j in range(env.n)])

print("\nOptimal Policy:")
for i in range(env.n):
    row = []
```

```
        for j in range(env.n):
            if (i, j) in env.terminal_states:
                row.append("T")
            else:
                row.append(list(optimal_policy[(i, j)].keys()))
        print(row)
```

Output:

```
Optimal Value Function:
[0, -1.0, -2.0, -3.0]
[-1.0, -2.0, -3.0, -2.0]
[-2.0, -3.0, -2.0, -1.0]
[-3.0, -2.0, -1.0, 0]

Optimal Policy:
['T', ['L'], ['L'], ['D', 'L']]
[['U'], ['U', 'L'], ['U', 'D', 'L', 'R'], ['D']]
[['U'], ['U', 'D', 'L', 'R'], ['D', 'R'], ['D']]
[['U', 'R'], ['R'], ['R'], 'T']
```

Learning Outcome:

Experiment – 5

Aim: Image Classification on MNSIT dataset (CNN Model with fully connected layers)

Theory:

1. Introduction

Image classification is a fundamental problem in computer vision where the goal is to assign a label to an input image. Convolutional Neural Networks (CNNs) are the **state-of-the-art deep learning models** for image classification tasks due to their ability to automatically extract spatial features from images.
In this experiment, a CNN model is implemented on the **MNIST dataset**, which consists of **28×28 grayscale images of handwritten digits (0–9)**. The model uses **convolutional layers** for feature extraction and **fully connected layers** for classification.

2. Convolutional Neural Networks (CNNs)

CNNs are designed to process grid-like data (images) and consist of three main types of layers:

(a) Convolutional Layers

- Perform **convolution operations** to extract local features like edges, textures, and patterns.
- Each convolution operation uses a **filter/kernel** that slides over the input image.
- The output is a **feature map** representing the presence of specific features in the image.

Mathematical operation:
where I is the input image and K is the kernel.

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n) \cdot K(m,n)$$

(b) Pooling Layers

- Reduce the spatial dimensions of feature maps to **decrease computation** and **increase feature robustness**.
- **Max pooling** and **average pooling** are commonly used.

Example:

- 2×2 max pooling selects the **maximum value** in each 2×2 window.

(c) Fully Connected Layers

- Flatten the feature maps and feed them into **dense layers**.
- Perform high-level reasoning and classification based on extracted features.

- The final layer typically uses a **softmax activation** for multi-class classification.

3. MNIST Dataset

- **Dataset size:** 70,000 images (60,000 for training, 10,000 for testing).
- **Image size:** 28×28 pixels (grayscale).
- **Classes:** 10 digits (0–9).

Preprocessing steps typically include:

- Normalization (scaling pixel values to 0–1).
- One-hot encoding of labels.

4. CNN Architecture for MNIST

A simple CNN for MNIST may include:

1. **Input layer:** 28×28 grayscale images.
2. **Convolutional layer 1:** 32 filters, 3×3 kernel, ReLU activation.
3. **Pooling layer 1:** 2×2 max pooling.
4. **Convolutional layer 2:** 64 filters, 3×3 kernel, ReLU activation.
5. **Pooling layer 2:** 2×2 max pooling.
6. **Flatten layer:** Converts 2D feature maps into 1D vector.
7. **Fully connected (dense) layer:** 128 neurons, ReLU activation.
8. **Output layer:** 10 neurons, softmax activation for classification.

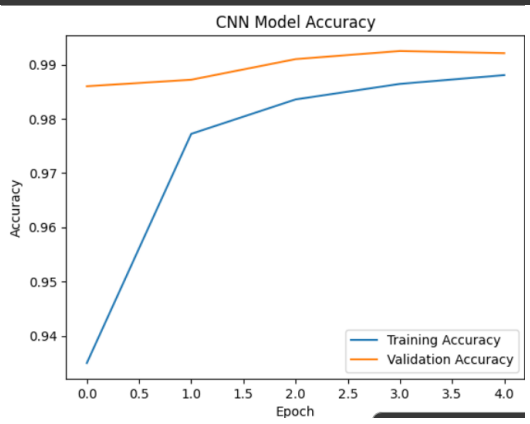
Code :

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt

# 1. Load and preprocess MNIST dataset
(x_train, y_train), (x_test, y_test) = datasets.mnist.load_data()

# Normalize pixel values to [0,1]
x_train, x_test = x_train / 255.0, x_test / 255.0

# Reshape data to include channel dimension (28x28x1)
x_train = x_train.reshape((-1, 28, 28, 1))
x_test = x_test.reshape((-1, 28, 28, 1))

# 2. Build CNN Model
model = models.Sequential([
    # Convolutional layer 1
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),

    # Convolutional layer 2
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    # Flatten the output before fully connected layers
    layers.Flatten(),

    # Fully connected layer 1
    layers.Dense(128, activation='relu'),
```

```
    # Dropout for regularization
    layers.Dropout(0.5),

    # Fully connected output layer
    layers.Dense(10, activation='softmax')
])

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

# 4. Train the model
history = model.fit(x_train, y_train, epochs=5,
                    validation_data=(x_test, y_test))

# 5. Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f"\nTest Accuracy: {test_acc:.4f}")

# 6. Plot training & validation accuracy
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('CNN Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

Output:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
/usr/local/lib/python3.12/dist-packages/keras/src/layers/convolutional/base_conv.py:113: UserWarning: Do not pass an `input_
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 36s 19ms/step - accuracy: 0.8630 - loss: 0.4316 - val_accuracy: 0.9860 - val_loss: 0.0438
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 38s 20ms/step - accuracy: 0.9757 - loss: 0.0828 - val_accuracy: 0.9872 - val_loss: 0.0367
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 38s 20ms/step - accuracy: 0.9828 - loss: 0.0581 - val_accuracy: 0.9910 - val_loss: 0.0277
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 38s 20ms/step - accuracy: 0.9858 - loss: 0.0469 - val_accuracy: 0.9925 - val_loss: 0.0244
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 36s 19ms/step - accuracy: 0.9881 - loss: 0.0385 - val_accuracy: 0.9921 - val_loss: 0.0255
313/313 - 2s - 7ms/step - accuracy: 0.9921 - loss: 0.0255

Test Accuracy: 0.9921
```



Learning Outcome:

Experiment – 6

Aim: Sentiment Analysis on IMDB dataset using RNN (LSTM)

Theory:

1. Introduction

Sentiment analysis is a natural language processing (NLP) task that involves identifying and classifying the **emotional tone** of text (e.g., positive, negative, or neutral).
Recurrent Neural Networks (RNNs), particularly **Long Short-Term Memory (LSTM)** networks, are well-suited for sequence-based data like text because they can **capture temporal dependencies** and context across sequences of words.
In this experiment, an **LSTM-based RNN model** is implemented to classify movie reviews in the **IMDB dataset** into positive or negative sentiment.

2. Theoretical Background

(a) Recurrent Neural Networks (RNNs)

RNNs are neural networks designed for **sequential data**. Unlike feedforward networks, RNNs have **loops** that allow information to persist across time steps:

$$h_t = f(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

Where:

- ht = hidden state at time ttt
- xt = input at time ttt
- Wxh,,Whh = weight matrices
- f = activation function (typically tanh or ReLU)

However, standard RNNs struggle with **long-term dependencies** due to vanishing or exploding gradients.

(b) Long Short-Term Memory (LSTM)

LSTMs are a type of RNN designed to overcome these limitations by introducing **memory cells** and **gating mechanisms**:

1. **Forget gate** – decides which information to discard from the cell state.
2. **Input gate** – decides which new information to store.
3. **Output gate** – decides what information to output.

This allows LSTMs to **retain relevant context** over long sequences, making them ideal for text-based tasks like sentiment analysis.

(c) IMDB Dataset

- Dataset contains **50,000 movie reviews**, equally divided into **positive and negative reviews**.

- Typically split into **25,000 training** and **25,000 testing** samples.
- Reviews are preprocessed into **sequences of integers**, representing word indices, often padded to a fixed length for uniform input.

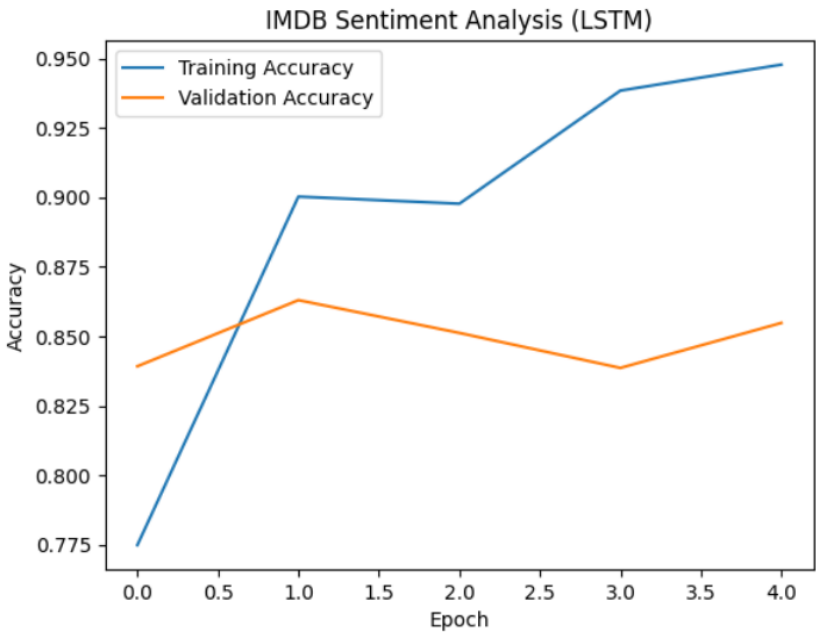
Code :

```
import tensorflow as tf
from tensorflow.keras import layers, models, datasets, preprocessing
import matplotlib.pyplot as plt

# 1. Load IMDB dataset (already pre-tokenized into integers)
# Keep only top 10,000 most frequent words
num_words = 10000
maxlen = 200  # limit each review to 200 words

(x_train, y_train), (x_test, y_test) = datasets.imdb.load_data(num_words=num_words)

# 2. Pad sequences to ensure equal length
x_train = preprocessing.sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = preprocessing.sequence.pad_sequences(x_test, maxlen=maxlen)

# 3. Build RNN Model with LSTM layers
model = models.Sequential([
    layers.Embedding(input_dim=num_words, output_dim=128, input_length=maxlen),
    layers.LSTM(128, return_sequences=False),   # You can replace with GRU(128)
    layers.Dropout(0.5),
    layers.Dense(64, activation='relu'),
    layers.Dense(1, activation='sigmoid')  # Binary sentiment: Positive or Negative
])

# 4. Compile model
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# 5. Train model
history = model.fit(x_train, y_train,
                    epochs=5,
                    batch_size=64,
                    validation_split=0.2)

# 6. Evaluate model
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f"\nTest Accuracy: {test_acc:.4f}")

# 7. Plot accuracy over epochs
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('IMDB Sentiment Analysis (LSTM)')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
```

```
plt.show()
```

Output:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz
17464789/17464789 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Epoch 1/5
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/embedding.py:97: UserWarning: Argument `input_length` is de
  warnings.warn(
313/313 ━━━━━━━━━━━━━━━━━━━━ 111s 348ms/step - accuracy: 0.6771 - loss: 0.5648 - val_accuracy: 0.8392 - val_loss: 0.3603
Epoch 2/5
313/313 ━━━━━━━━━━━━━━━━━━━━ 108s 344ms/step - accuracy: 0.9026 - loss: 0.2491 - val_accuracy: 0.8630 - val_loss: 0.3264
Epoch 3/5
313/313 ━━━━━━━━━━━━━━━━━━━━ 136s 325ms/step - accuracy: 0.9136 - loss: 0.2227 - val_accuracy: 0.8512 - val_loss: 0.4037
Epoch 4/5
313/313 ━━━━━━━━━━━━━━━━━━━━ 104s 333ms/step - accuracy: 0.9369 - loss: 0.1641 - val_accuracy: 0.8386 - val_loss: 0.3959
Epoch 5/5
313/313 ━━━━━━━━━━━━━━━━━━━━ 108s 345ms/step - accuracy: 0.9603 - loss: 0.1116 - val_accuracy: 0.8548 - val_loss: 0.4337
782/782 - 43s - 56ms/step - accuracy: 0.8552 - loss: 0.4289

Test Accuracy: 0.8552
```



Learning Outcome:

Experiment – 7

Aim: Transfer Learning using Pre-trained CNN Models on the **CIFAR-10 dataset**.

Theory:

1. Introduction

Transfer learning is a machine learning technique where a model **pre-trained on a large dataset** is adapted for a different but related task.
Instead of training a deep learning model from scratch, transfer learning allows **leveraging previously learned features** to save computational resources and achieve faster convergence.
In this experiment, **pre-trained CNN models** such as **VGG16, ResNet50, or InceptionV3** are applied to the **CIFAR-10 dataset**, which consists of **32×32 color images across 10 classes**.

2. Theoretical Background

(a) Transfer Learning

Transfer learning is based on the idea that **early layers of CNNs learn generic features** like edges, textures, and shapes, which are often applicable to many vision tasks.
Two main approaches:

1. **Feature extraction:** Use a pre-trained model as a **fixed feature extractor**; only train the classifier layers.
2. **Fine-tuning:** Unfreeze some layers of the pre-trained model and **retrain them** on the new dataset for better adaptation.

(b) Pre-trained CNN Models

Popular pre-trained CNNs include:

- **VGG16/VGG19:** Deep convolutional network with 16/19 layers.
- **ResNet50:** Introduces **residual connections** to enable very deep networks without vanishing gradients.
- **InceptionV3:** Uses **parallel convolutional filters** to capture multi-scale features efficiently.

These models are trained on **ImageNet** (over 1 million images, 1000 classes) and can generalize to smaller datasets like CIFAR-10.

(c) CIFAR-10 Dataset

- **Dataset size:** 60,000 images (50,000 train, 10,000 test)
- **Image size:** 32×32 pixels, RGB
- **Classes:** 10 categories (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck)

Preprocessing steps:

- Resize images to fit the input size of the pre-trained model (e.g., 224×224 for VGG16).
- Normalize pixel values (0–1 or -1 to 1).
- Encode labels as one-hot vectors.

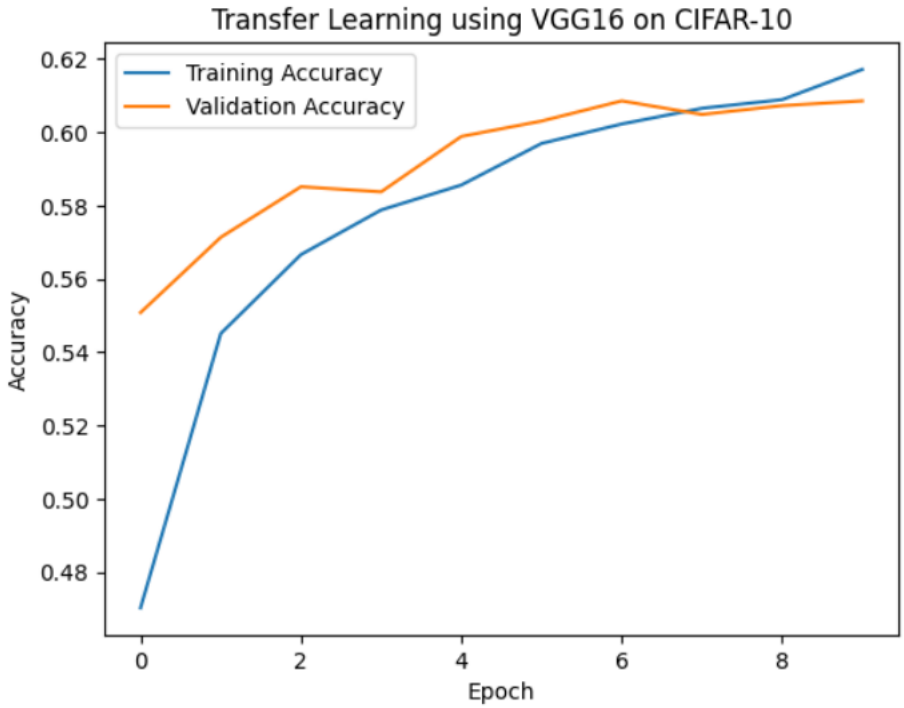
Code:

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.applications import VGG16
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt

# 1. Load CIFAR-10 dataset (10 classes, 32x32 color images)
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize image pixel values
x_train, x_test = x_train / 255.0, x_test / 255.0

# Convert labels to one-hot encoding
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# 2. Load pre-trained VGG16 model (without top layers)
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(32, 32, 3))

# Freeze all convolutional layers (so their weights won't change during training)
for layer in base_model.layers:
    layer.trainable = False

# 3. Add custom fully connected layers for CIFAR-10 classification
model = models.Sequential([
    base_model,
    layers.Flatten(),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(10, activation='softmax')  # 10 output classes
])

# 4. Compile model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# 5. Train model
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=64,
                    validation_data=(x_test, y_test))

# 6. Evaluate model
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f"\nTest Accuracy: {test_acc:.4f}")

# 7. Plot training and validation accuracy
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Transfer Learning using VGG16 on CIFAR-10')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
```

```
plt.legend()
plt.show()
```

Output:

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 3s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf
58889256/58889256 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Epoch 1/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 715s 912ms/step - accuracy: 0.3999 - loss: 1.6982 - val_accuracy: 0.5508 - val_loss: 1.3037
Epoch 2/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 729s 933ms/step - accuracy: 0.5447 - loss: 1.3009 - val_accuracy: 0.5713 - val_loss: 1.2199
Epoch 3/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 722s 923ms/step - accuracy: 0.5660 - loss: 1.2421 - val_accuracy: 0.5851 - val_loss: 1.1969
Epoch 4/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 743s 924ms/step - accuracy: 0.5790 - loss: 1.2029 - val_accuracy: 0.5837 - val_loss: 1.1718
Epoch 5/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 723s 925ms/step - accuracy: 0.5863 - loss: 1.1770 - val_accuracy: 0.5988 - val_loss: 1.1510
Epoch 6/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 723s 924ms/step - accuracy: 0.5948 - loss: 1.1595 - val_accuracy: 0.6030 - val_loss: 1.1385
Epoch 7/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 742s 924ms/step - accuracy: 0.6049 - loss: 1.1268 - val_accuracy: 0.6085 - val_loss: 1.1238
Epoch 8/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 725s 928ms/step - accuracy: 0.6093 - loss: 1.1144 - val_accuracy: 0.6048 - val_loss: 1.1307
Epoch 9/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 725s 928ms/step - accuracy: 0.6109 - loss: 1.1077 - val_accuracy: 0.6072 - val_loss: 1.1172
Epoch 10/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 713s 891ms/step - accuracy: 0.6180 - loss: 1.0904 - val_accuracy: 0.6085 - val_loss: 1.1203
313/313 - 117s - 374ms/step - accuracy: 0.6085 - loss: 1.1203

Test Accuracy: 0.6085
```



Learning Outcome:

Experiment – 8

Aim: Image Caption Generator using CNN + LSTM

Theory:

1. Introduction

An image caption generator is an AI system that generates natural language descriptions for images. This task combines computer vision and natural language processing.
The typical architecture uses:

- CNNs to extract image features.
- LSTMs (RNNs) to generate sequences of words forming a descriptive caption.

This experiment implements a CNN + LSTM model to generate captions for images from datasets like Flickr8k or MS COCO.

2. Theoretical Background

(a) Convolutional Neural Networks (CNNs)

- CNNs extract **spatial features** from images.
- Feature maps from CNN layers act as a **vector representation** of the image content.
- Common pre-trained CNNs for feature extraction include **VGG16, InceptionV3, and ResNet50**.

(b) Recurrent Neural Networks (RNNs) and LSTMs

- LSTMs are used for **sequence generation**, handling **variable-length outputs** like sentences.
- LSTM uses **memory cells** and **gates** (input, forget, output) to maintain context across sequences.
- In image captioning, LSTM predicts the **next word in the caption** based on the image features and previously generated words.

Mathematical operation for LSTM step:

$$h_t = \text{LSTM}(x_t, h_{t-1})$$

Where xt is the input at time step ttt (previous word embedding), and h{t-1} is the previous hidden state.

(c) CNN + LSTM Architecture

1. **Feature Extraction:** Pass the image through a CNN (without top layers) to get feature vector.
2. **Embedding Layer:** Map each word in the vocabulary to a dense vector.
3. **Sequence Model (LSTM):** Generate captions word-by-word using the previous words and image features.
4. **Output Layer:** Softmax layer over vocabulary to predict the next word.

(d) Dataset

Common datasets for image captioning:

- **Flickr8k/Flickr30k:** Thousands of images with multiple captions each.
- **MS COCO:** 120,000 images with five captions per image.

Preprocessing includes:

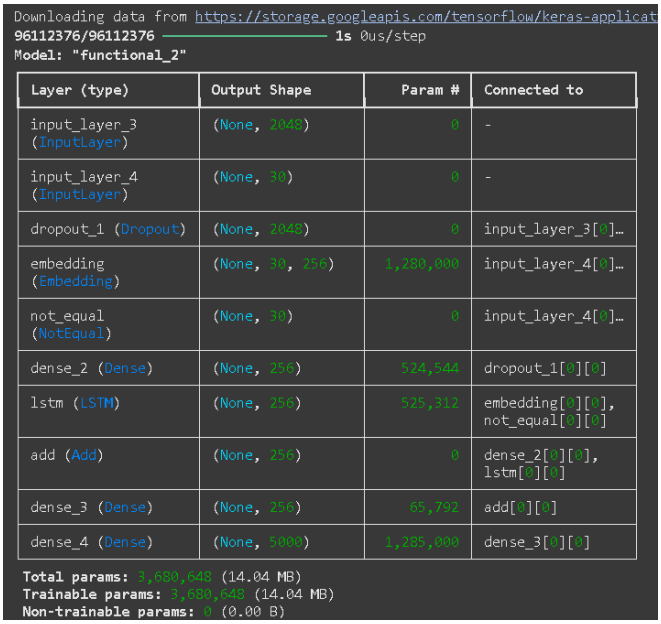
- Tokenizing and padding captions.
- Normalizing images for CNN input.
- Splitting data into training and validation sets.

Code:

```
# Image Caption Generator using CNN + LSTM
import tensorflow as tf
from tensorflow.keras import layers, models, Input
from tensorflow.keras.applications import InceptionV3
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical
import numpy as np

# ----------------------------------------------------------
# 1. CNN Model (Feature Extractor) - Pre-trained InceptionV3
# ----------------------------------------------------------
cnn_model = InceptionV3(weights='imagenet')
cnn_model = models.Model(cnn_model.input, cnn_model.layers[-2].output)

def extract_features(image_path):
    """Extract features from image using pretrained CNN"""
    from tensorflow.keras.preprocessing import image
    img = image.load_img(image_path, target_size=(299, 299))
    img = image.img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = tf.keras.applications.inception_v3.preprocess_input(img)
    features = cnn_model.predict(img, verbose=0)
    return features

# Example (replace with actual image path)
# features = extract_features('sample.jpg')

# ----------------------------------------------------------
# 2. Text Preprocessing & Vocabulary Setup
# ----------------------------------------------------------
# Example vocabulary and captions (for demo)
vocab_size = 5000
max_length = 30

# Dummy encoded captions and features (for demonstration)
X1 = np.random.randn(1000, 2048)          # Image features from CNN
X2 = np.random.randint(1, vocab_size, (1000, max_length))  # Input text sequence
y = np.random.randint(1, vocab_size, (1000, 1))           # Next word prediction

# ----------------------------------------------------------
# 3. Define the Combined CNN + LSTM Model
# ----------------------------------------------------------
# Feature extractor (CNN part)
inputs1 = Input(shape=(2048,))
fe1 = layers.Dropout(0.5)(inputs1)
fe2 = layers.Dense(256, activation='relu')(fe1)

# Sequence processor (LSTM part)
inputs2 = Input(shape=(max_length,))
se1 = layers.Embedding(vocab_size, 256, mask_zero=True)(inputs2)
```

```
se2 = layers.LSTM(256)(se1)

# Decoder (merge image and sequence)
decoder1 = layers.add([fe2, se2])
decoder2 = layers.Dense(256, activation='relu')(decoder1)
outputs = layers.Dense(vocab_size, activation='softmax')(decoder2)

model = models.Model(inputs=[inputs1, inputs2], outputs=outputs)
model.compile(loss='categorical_crossentropy', optimizer='adam')

model.summary()

# ------------------------------------------------------------
# 4. Train the model (example training loop)
# ------------------------------------------------------------
# Convert y to one-hot encoded form
y_onehot = to_categorical(y, num_classes=vocab_size)

history = model.fit([X1, X2], y_onehot, epochs=5, batch_size=32, verbose=1)

# ------------------------------------------------------------
# 5. Caption Generation (Inference)
# ------------------------------------------------------------
def generate_caption(model, photo_features, tokenizer, max_length=30):
    """Generate caption for image features"""
    in_text = 'startseq'
    for _ in range(max_length):
        sequence = tokenizer.texts_to_sequences([in_text])[0]
        sequence = pad_sequences([sequence], maxlen=max_length)
        yhat = np.argmax(model.predict([photo_features, sequence], verbose=0))
        word = tokenizer.index_word.get(yhat, None)
        if word is None:
            break
        in_text += ' ' + word
        if word == 'endseq':
            break
    return in_text

print("\nModel ready for caption generation!")
```

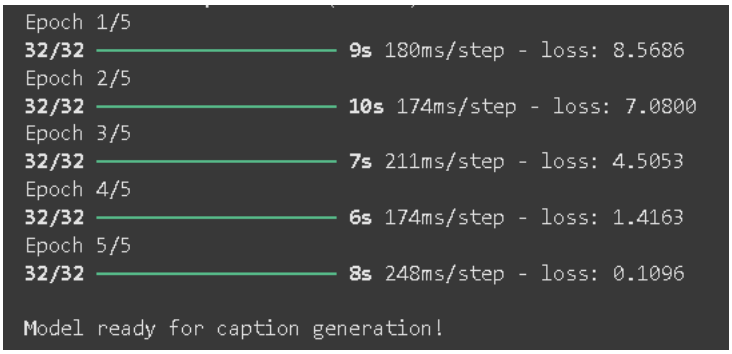
Output:



Downloading data from https://storage.googleapis.com/tensorflow/keras-applicat
96112376/96112376 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
Model: "functional_2"

Layer (type)	Output Shape	Param #	Connected to
input_layer_3 (InputLayer)	(None, 2048)	0	-
input_layer_4 (InputLayer)	(None, 30)	0	-
dropout_1 (Dropout)	(None, 2048)	0	input_layer_3[0]…
embedding (Embedding)	(None, 30, 256)	1,280,000	input_layer_4[0]…
not_equal (NotEqual)	(None, 30)	0	input_layer_4[0]…
dense_2 (Dense)	(None, 256)	524,544	dropout_1[0][0]
lstm (LSTM)	(None, 256)	525,312	embedding[0][0], not_equal[0][0]
add (Add)	(None, 256)	0	dense_2[0][0], lstm[0][0]
dense_3 (Dense)	(None, 256)	65,792	add[0][0]
dense_4 (Dense)	(None, 5000)	1,285,000	dense_3[0][0]

Total params: 3,680,648 (14.04 MB)
Trainable params: 3,680,648 (14.04 MB)
Non-trainable params: 0 (0.00 B)



```
Epoch 1/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 9s 180ms/step - loss: 8.5686
Epoch 2/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 10s 174ms/step - loss: 7.0800
Epoch 3/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 7s 211ms/step - loss: 4.5053
Epoch 4/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 6s 174ms/step - loss: 1.4163
Epoch 5/5
32/32 ━━━━━━━━━━━━━━━━━━━━ 8s 248ms/step - loss: 0.1096

Model ready for caption generation!
```

Learning Outcome:

Experiment – 9

Aim: Face Mask Detection using CNN.

Theory:

1. Introduction

Face mask detection is a **computer vision task** aimed at identifying whether a person is **wearing a mask** or **not wearing a mask** in an image or video stream.
This task has become particularly important in **public health monitoring**, especially during pandemics like COVID-19.
Convolutional Neural Networks (CNNs) are widely used for face mask detection due to their **ability to automatically extract hierarchical features** from images.

2. Theoretical Background

(a) Convolutional Neural Networks (CNNs)

CNNs are deep learning models specialized for **image processing tasks**.

- **Convolutional layers:** Extract features such as edges, textures, and patterns.
- **Pooling layers:** Reduce spatial dimensions, retaining essential features.
- **Fully connected layers:** Perform classification based on extracted features.

The typical CNN architecture for mask detection includes multiple convolution and pooling layers followed by fully connected layers with **softmax or sigmoid activation** for classification.

(b) Dataset

A face mask detection dataset usually contains **images of faces with and without masks**, labeled as:

- **Mask (1)**
- **No Mask (0)**

Preprocessing steps:

1. **Resize images** to a uniform size (e.g., 224×224 pixels).
2. **Normalize pixel values** (0–1).
3. **Augment data** (rotation, flipping, zoom) to improve generalization.

(c) CNN Architecture for Mask Detection

1. **Input layer:** Accepts preprocessed images.
2. **Convolution + ReLU layers:** Extract hierarchical features.
3. **Max pooling layers:** Downsample feature maps.
4. **Flatten layer:** Converts feature maps into a vector.
5. **Fully connected layer(s):** Classifies the image as mask or no-mask.
6. **Output layer:** Sigmoid activation for binary classification.

Code:

```
# Create dataset folder
!mkdir -p dataset

# Download a sample face mask dataset (small and easy to use)
```

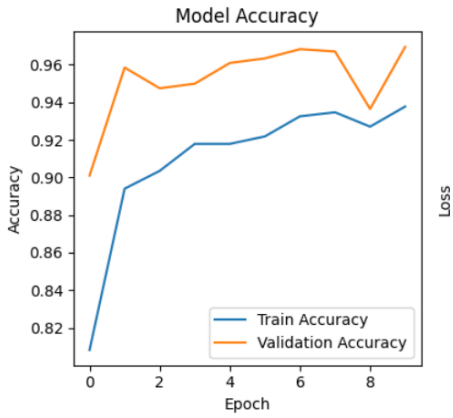
```
!wget -q https://github.com/chandrikadeb7/Face-Mask-Detection/archive/refs/heads/master.zip -O
face_mask.zip

# Unzip into dataset folder
!unzip -q face_mask.zip -d dataset

# ------------------------------------------------------------
# FACE MASK DETECTION USING CNN
# ------------------------------------------------------------

import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.keras.preprocessing import image
import os

# ------------------------------------------------------------
# 2 ☐ Dataset Preparation
# ------------------------------------------------------------
dataset_dir = 'dataset/Face-Mask-Detection-master/dataset'

# Use ImageDataGenerator with validation split (80% train, 20% validation)
datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    zoom_range=0.2,
    shear_range=0.2,
    horizontal_flip=True,
    validation_split=0.2
)

train_data = datagen.flow_from_directory(
    dataset_dir,
    target_size=(128, 128),
    batch_size=32,
    class_mode='binary',
    subset='training'
)

val_data = datagen.flow_from_directory(
    dataset_dir,
    target_size=(128, 128),
    batch_size=32,
    class_mode='binary',
    subset='validation'
)

# ------------------------------------------------------------
# 3 ☐ Build CNN Model
# ------------------------------------------------------------
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 3)),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
```

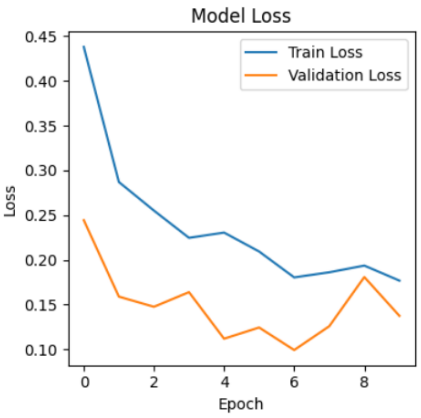
```
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(1, activation='sigmoid')  # Binary output: Mask / No Mask
])

# ------------------------------------------------------------
# 4 ☐ Compile Model
# ------------------------------------------------------------
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# ------------------------------------------------------------
# 5 ☐ Train Model
# ------------------------------------------------------------
history = model.fit(
    train_data,
    epochs=10,
    validation_data=val_data
)

# ------------------------------------------------------------
# 6 ☐ Evaluate Model
# ------------------------------------------------------------
test_loss, test_acc = model.evaluate(val_data, verbose=2)
print(f"\nValidation Accuracy: {test_acc:.4f}")

# ------------------------------------------------------------
# 7 ☐ Plot Accuracy and Loss
# ------------------------------------------------------------
plt.figure(figsize=(8, 4))

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()

# ------------------------------------------------------------
# 8 ☐ Predict on a Single Image
# ------------------------------------------------------------
def predict_mask(img_path):
    img = image.load_img(img_path, target_size=(128, 128))
    img_array = image.img_to_array(img)
    img_array = np.expand_dims(img_array, axis=0) / 255.0
    prediction = model.predict(img_array)[0][0]

    plt.imshow(img)
```

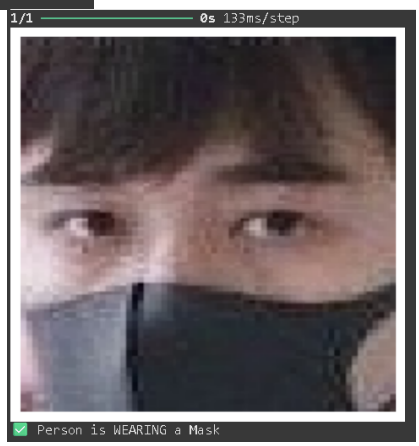
```
    plt.axis('off')
    plt.show()

    if prediction < 0.5:
        print("✅ Person is WEARING a Mask")
    else:
        print("❌ Person is NOT WEARING a Mask")

# Example usage — use one image from dataset:
example_image = os.path.join(dataset_dir, "with_mask", os.listdir(os.path.join(dataset_dir, "with_mask"))[0])
predict_mask(example_image)
```

Output:

```
Found 3274 images belonging to 2 classes.
Found 818 images belonging to 2 classes.
/usr/local/lib/python3.12/dist-packages/keras/src/layers/convolutional/base_conv.py:113: UserWarning: Do not pass an `i
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
/usr/local/lib/python3.12/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121: UserWarning: Your `
  self._warn_if_super_not_called()
Epoch 1/10
 62/103 ━━━━━━━━━━━━━━━━━━━━ 50s 1s/step - accuracy: 0.6432 - loss: 0.6221/usr/local/lib/python3.12/dist-packages/PIL/I
  warnings.warn(
103/103 ━━━━━━━━━━━━━━━━━━━━ 140s 1s/step - accuracy: 0.7013 - loss: 0.5590 - val_accuracy: 0.9010 - val_loss: 0.2443
Epoch 2/10
103/103 ━━━━━━━━━━━━━━━━━━━━ 146s 1s/step - accuracy: 0.8900 - loss: 0.3066 - val_accuracy: 0.9584 - val_loss: 0.1589
Epoch 3/10
103/103 ━━━━━━━━━━━━━━━━━━━━ 134s 1s/step - accuracy: 0.9042 - loss: 0.2565 - val_accuracy: 0.9474 - val_loss: 0.1475
Epoch 4/10
103/103 ━━━━━━━━━━━━━━━━━━━━ 136s 1s/step - accuracy: 0.9147 - loss: 0.2342 - val_accuracy: 0.9499 - val_loss: 0.1639
Epoch 5/10
103/103 ━━━━━━━━━━━━━━━━━━━━ 135s 1s/step - accuracy: 0.9220 - loss: 0.2288 - val_accuracy: 0.9609 - val_loss: 0.1118
Epoch 6/10
103/103 ━━━━━━━━━━━━━━━━━━━━ 135s 1s/step - accuracy: 0.9181 - loss: 0.2187 - val_accuracy: 0.9633 - val_loss: 0.1243
Epoch 7/10
103/103 ━━━━━━━━━━━━━━━━━━━━ 134s 1s/step - accuracy: 0.9261 - loss: 0.1927 - val_accuracy: 0.9682 - val_loss: 0.0991
Epoch 8/10
103/103 ━━━━━━━━━━━━━━━━━━━━ 133s 1s/step - accuracy: 0.9365 - loss: 0.1643 - val_accuracy: 0.9670 - val_loss: 0.1257
Epoch 9/10
103/103 ━━━━━━━━━━━━━━━━━━━━ 129s 1s/step - accuracy: 0.9302 - loss: 0.1968 - val_accuracy: 0.9364 - val_loss: 0.1807
Epoch 10/10
103/103 ━━━━━━━━━━━━━━━━━━━━ 131s 1s/step - accuracy: 0.9370 - loss: 0.1801 - val_accuracy: 0.9694 - val_loss: 0.1374
26/26 - 11s - 427ms/step - accuracy: 0.9621 - loss: 0.1310

Validation Accuracy: 0.9621
```







Learning Outcome:

Experiment – 10

Aim: Named Entity Recognition (NER) using Bi-LSTM + CRF.

Theory:

1. Introduction

Named Entity Recognition (NER) is a fundamental task in **Natural Language Processing (NLP)** that identifies and classifies entities in text into predefined categories such as:

- **Person names**
- **Locations**
- **Organizations**
- **Dates, monetary values, etc.**

NER is widely used in **information extraction, question answering, and knowledge graph construction**.

This experiment implements NER using a combination of **Bidirectional LSTM (Bi-LSTM)** networks and **Conditional Random Fields (CRF)** for sequence labeling.

2. Theoretical Background

(a) Bidirectional LSTM (Bi-LSTM)

- Standard LSTM processes sequences in **one direction** (past → future).
- Bi-LSTM processes sequences in **both directions** (past → future and future → past), providing **context from both sides** for each token.

Mathematical representation for a token at position t:

$$\overrightarrow{h_t} = \text{LSTM}_{\text{forward}}(x_t, h_{t-1})$$

$$\overleftarrow{h_t} = \text{LSTM}_{\text{backward}}(x_t, h_{t+1})$$

$$h_t = [\overrightarrow{h_t}; \overleftarrow{h_t}]$$

Where ht is the concatenated hidden state combining forward and backward information.

(b) Conditional Random Fields (CRF)

- CRF is a probabilistic **sequence modeling algorithm**.
- It captures **dependencies between labels**, enforcing valid label sequences (e.g., an "I-PER" tag must follow a "B-PER").
- Using CRF on top of Bi-LSTM ensures **globally optimal label sequences** instead of predicting each token independently.

(c) Bi-LSTM + CRF Architecture for NER

1. **Input Layer:** Tokenized words converted to embeddings (e.g., Word2Vec or GloVe).
2. **Bi-LSTM Layer:** Generates contextualized representations for each token.
3. **CRF Layer:** Predicts the most likely sequence of entity tags.

This combination captures **both word-level context** and **label dependencies**, improving NER accuracy.

Code:

```
# Install keras-crf (works fine on Colab)
!pip install keras-crf

import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical
from keras_crf import CRF

# ---------------------------------------------------------

# 1. Sample Data (demo only — replace with your real dataset)

# ---------------------------------------------------------
sentences = [
    ["John", "lives", "in", "New", "York"],
    ["Mary", "works", "at", "Google"],
    ["India", "is", "a", "country"]
]

# Corresponding NER tags (BIO format)
tags = [
    ["B-PER", "O", "O", "B-LOC", "I-LOC"],
    ["B-PER", "O", "O", "B-ORG"],
    ["B-LOC", "O", "O", "O"]

]

# ---------------------------------------------------------

# 2. Vocabulary prep

# ---------------------------------------------------------

words = sorted({w for s in sentences for w in s})
tags_set = sorted({t for ts in tags for t in ts})
word2idx = {w: i + 2 for i, w in enumerate(words)}  # +2 for PAD and OOV
word2idx["PAD"] = 0
word2idx["OOV"] = 1
idx2tag = {i: t for i, t in enumerate(tags_set)}
tag2idx = {t: i for i, t in idx2tag.items()}

# ---------------------------------------------------------

# 3. Sequence prep

# ---------------------------------------------------------

X = [[word2idx.get(w, 1) for w in s] for s in sentences]
y = [[tag2idx[t] for t in ts] for ts in tags]
max_len = max(len(s) for s in X)
X = pad_sequences(X, maxlen=max_len, padding='post')
y = pad_sequences(y, maxlen=max_len, padding='post')
```

```
y = to_categorical(y, num_classes=len(tags_set))

# ------------------------------------------------------------

# 4. Build BiLSTM + CRF model

# ------------------------------------------------------------

input_word = layers.Input(shape=(max_len,))
emb = layers.Embedding(input_dim=len(word2idx), output_dim=64,
input_length=max_len)(input_word)
bilstm = layers.Bidirectional(layers.LSTM(units=64, return_sequences=True))(emb)
dense = layers.TimeDistributed(layers.Dense(64, activation="relu"))(bilstm)
crf = CRF(len(tags_set))
output = crf(dense)
ner_model = models.Model(input_word, output)
ner_model.compile(optimizer='adam',
                  loss=crf.loss_function,
                  metrics=[crf.accuracy])
ner_model.summary()

# ------------------------------------------------------------

# 5. Train model

# ------------------------------------------------------------

history = ner_model.fit(X, y, batch_size=2, epochs=15, verbose=1)

# ------------------------------------------------------------

# 6. Evaluate / predict

# ------------------------------------------------------------

pred = ner_model.predict(X)
pred_tags = np.argmax(pred, axis=-1)
for i, sentence in enumerate(sentences):
    print(f"\nSentence: {' '.join(sentence)}")
    print("Predicted Tags:",
          [idx2tag[idx] for idx in pred_tags[i][:len(sentence)]])
```

Output:

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 5)]               0

 embedding (Embedding)       (None, 5, 64)             768

 bidirectional (Bidirectio   (None, 5, 128)            66048
 nal)

 time_distributed (TimeDis   (None, 5, 64)             8256
 tributed)

 crf (CRF)                   (None, 5, 8)              592

=================================================================
Total params: 76,264
Trainable params: 76,264
Non-trainable params: 0
```

```
Epoch 1/15
2/2 [==============================] - 5s 15ms/step - loss: 1.6369 - crf_accuracy: 0.2667
Epoch 2/15
2/2 [==============================] - 0s 6ms/step - loss: 1.5866 - crf_accuracy: 0.3333
Epoch 3/15
2/2 [==============================] - 0s 6ms/step - loss: 1.5284 - crf_accuracy: 0.4000
```

```
1/1 [==============================] - 0s 458ms/step

Sentence: John lives in New York
Predicted Tags: ['B-PER', 'O', 'O', 'B-LOC', 'I-LOC']

Sentence: Mary works at Google
Predicted Tags: ['B-PER', 'O', 'O', 'B-ORG']

Sentence: India is a country
Predicted Tags: ['B-LOC', 'O', 'O', 'O']
```

Learning Outcome: