EXPERIMENT - 3

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

- **Environment set up and intro to OpenAI Gym**
- **Write Q-learning algorithm and train agent to play game**
- **Watch trained agent play game**

THEORY:

One of the most widely used algorithms in RL is **Q-Learning**, a model-free, off-policy method that enables an agent to learn the optimal policy through trial and error without needing a model of the environment.

Environment and OpenAI Gym:
To train and test RL algorithms, we require a standardized simulation environment. **OpenAI Gym** is a popular Python library that provides a wide variety of environments—from simple grid worlds to complex games—for testing reinforcement learning algorithms. Each environment consists of:

- **States (S):** The current configuration or observation of the environment.
- **Actions (A):** The possible moves the agent can make.
- **Rewards (R):** Numerical feedback received after performing an action.
- **Episodes:** A sequence of state transitions until a terminal condition (win/loss/end) is reached.

In this setup, the agent interacts with the environment in a loop: it observes the state, takes an action, receives a reward, and moves to a new state.

Q-Learning Algorithm:
Q-Learning is based on the **Bellman Equation**, which iteratively updates a Q-value (quality) function, (Q(s,a)), representing the expected cumulative reward of taking action *a* in state *s*, and then following the best policy. The update rule is:

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

where:

- (α) is the **learning rate** ($0 < \alpha \leq 1$) controlling how much new information overrides old knowledge.
- (γ) is the **discount factor** ($0 \leq \gamma \leq 1$) determining the importance of future rewards.
- (r) is the **reward** received after taking action *a* in state *s*.
- (s') is the **next state**.

The algorithm proceeds as follows:

1. Initialize Q-table with zeros.
2. For each episode:
    - Start from an initial state.
    - Choose an action using an **ε-greedy policy** (explore with probability ε, exploit with 1−ε).
    - Perform the action, observe the next state and reward.
    - Update the Q-value using the formula above.
    - Repeat until the episode ends.
3. Over many episodes, the Q-values converge to the optimal policy.

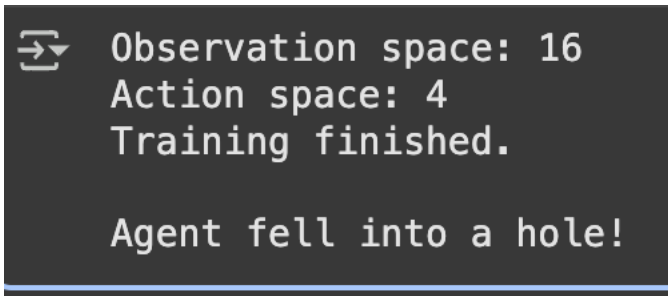
Training and Watching the Agent Play:
In Python, using OpenAI Gym (e.g., `gym.make('FrozenLake-v1')`), the Q-Learning algorithm can be implemented with basic libraries like NumPy. During training, the agent explores various actions and updates its Q-table. Once training is complete, the exploration rate (ε) is reduced, and the agent exploits its learned Q-values to select the best actions. Finally, the trained agent can be observed playing the game autonomously, demonstrating intelligent decision-making behavior based on the learned policy.

CODE:

```
import numpy as np

# Gym import compatible with both gymnasium and gym
try:
    import gymnasium as gym
    NEW_API = True
except Exception:
    import gym  # type: ignore
    NEW_API = False

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

n_states = env.observation_space.n
n_actions = env.action_space.n
print("Observation space:", n_states)
print("Action space:", n_actions)

# Q-learning hyperparameters
alpha = 0.8
gamma = 0.95
epsilon = 1.0
epsilon_decay = 0.995
epsilon_min = 0.01
episodes = 2000
max_steps = 100

q_table = np.zeros((n_states, n_actions))
```

```
for episode in range(episodes):
    if NEW_API:
        state, _ = env.reset()
    else:
        state = env.reset()
    done = False

    for _ in range(max_steps):
        # epsilon-greedy action
        if np.random.rand() < epsilon:
            action = env.action_space.sample()
        else:
            action = int(np.argmax(q_table[state]))

        if NEW_API:
            next_state, reward, terminated, truncated, _ =
env.step(action)
            done = terminated or truncated
        else:
            next_state, reward, done, _ = env.step(action)

        # Q-update
        q_table[state, action] = 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")

# Greedy run
if NEW_API:
    state, _ = env.reset()
else:
    state = env.reset()

for step in range(max_steps):
    action = int(np.argmax(q_table[state]))
    if NEW_API:
        next_state, reward, terminated, truncated, _ = env.step(action)
        done = terminated or truncated
    else:
        next_state, reward, done, _ = env.step(action)

    state = next_state
    if done:
        if reward > 0:
            print("Agent reached the goal!")
        else:
            print("Agent fell into a hole!")
        break
env.close()
```

OUTPUT:



```
Observation space: 16
Action space: 4
Training finished.

Agent fell into a hole!
```

LEARNING OUTCOME:

EXPERIMENT - 4

AIM: Python implementation of the iterative policy evaluation and update

THEORY:

In reinforcement learning, an **agent** interacts with an environment to learn optimal actions that maximize rewards. One of the foundational frameworks for understanding how agents learn optimal behavior is the **Markov Decision Process (MDP)**, which consists of states, actions, rewards, and transition probabilities. To find the optimal policy—the best mapping from states to actions—we use techniques such as **Iterative Policy Evaluation** and **Greedy Policy Improvement**. Together, these form the basis of the **Policy Iteration** algorithm.

Iterative Policy Evaluation:
A **policy** (denoted by π) is a strategy that tells the agent which action to take in each state. To evaluate how good a given policy is, we compute its **state-value function**, ($V^\pi(s)$), which represents the expected return when starting from state *s* and following policy π thereafter.

The value function can be defined using the **Bellman expectation equation**:

$$V^\pi(s) = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)[r + \gamma V^\pi(s')]$$

Here,

- ($\pi(a \mid s)$) is the probability of taking action *a* in state *s* under policy π.
- ($p(s', r \mid s, a)$) is the probability of transitioning to state *s*′ and receiving reward *r* after taking action *a* in state *s*.
- ($p(s', r \mid s, a)$) is the **discount factor**, which determines how much future rewards are valued.

Iterative Policy Evaluation repeatedly applies this equation to update the value of each state until the values converge (i.e., the change between updates becomes negligible). This iterative approach ensures that we accurately estimate how good a given policy is across all states in the environment.

Greedy Policy Improvement:
Once we have evaluated a policy, we can **improve** it by making it greedy with respect to the computed value function. This means that for each state, we choose the action that maximizes the expected return:

$$\pi'(s) = \arg\underset{a}{max} \sum_{s',r} p(s', r \mid s, a)[r + \gamma V^\pi(s')]$$

This new policy (\pi') is guaranteed to be at least as good as (and often better than) the old policy π. The process of **evaluating** a policy and then **improving** it continues iteratively until no further improvement is possible—at which point the policy has converged to the **optimal policy**, (\pi^*).

Gridworld Example:
A **Gridworld** is a simple environment used to illustrate policy evaluation and improvement. The environment consists of a grid of cells representing states. The agent can move up, down, left, or right. Some states may have rewards or penalties, and some are **terminal states**, where the episode ends.

For example, in a 4×4 grid:

- The goal state might provide a reward of +1.
- A terminal or trap state might give a reward of 0 or -1.
- All other moves yield a small negative reward (e.g., -0.04) to encourage shorter paths.

Using **iterative policy evaluation**, we first calculate the value of each state under a random policy. Then, through **greedy policy improvement**, the agent gradually refines its actions to move optimally toward the goal while avoiding traps.

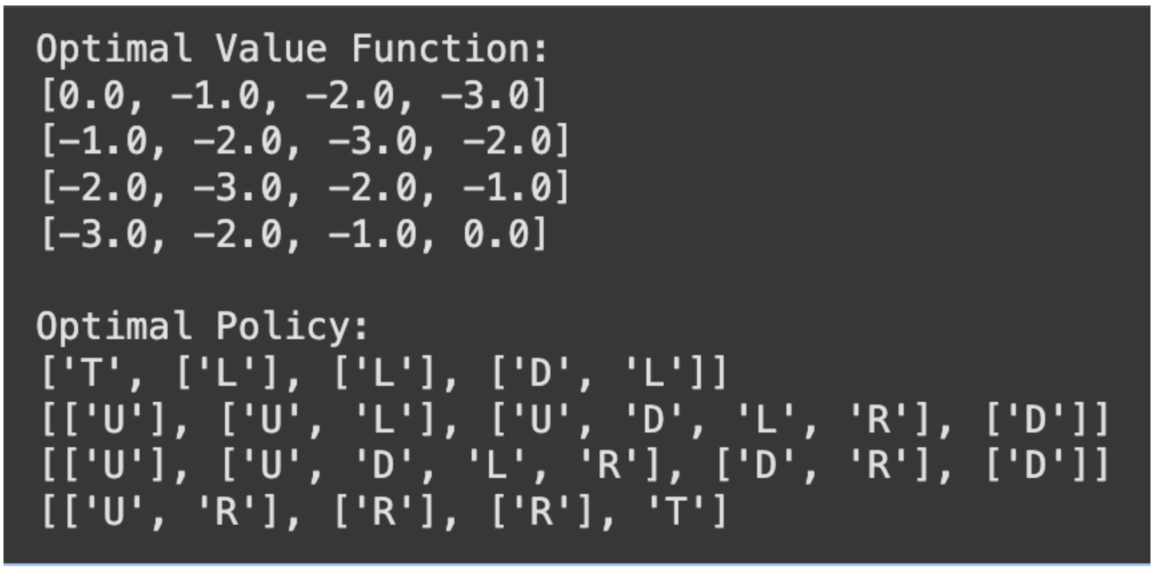
Conclusion:
Iterative policy evaluation and greedy policy improvement together form the foundation of **policy iteration**, a core method in reinforcement learning. The Gridworld example provides an intuitive way to visualize how an agent learns from feedback to optimize its path to a goal. Through repeated evaluation and improvement, the agent converges on an optimal policy that maximizes its expected cumulative reward.

CODE:

```
from __future__ import annotations
import math

class Gridworld:
    def __init__(self, n: int = 4, gamma: float = 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
        return next_state, reward
```

```
def policy_evaluation(env: Gridworld, policy, theta: float = 1e-6,
gamma: float = 1.0):
    V = {s: 0.0 for s in env.states}
    while True:
        delta = 0.0
        for state in env.states:
            if state in env.terminal_states:
                continue
            v = V[state]
            new_v = 0.0
            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

def policy_improvement(env: Gridworld, V, gamma: float = 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]
        max_q = max(q_values.values())
        best_actions = [a for a, q in q_values.items() if q == max_q]
        policy[state] = {a: 1.0 / len(best_actions) for a in
best_actions}
    return policy

def policy_iteration(env: Gridworld, gamma: float = 1.0, theta: float =
1e-6):
    policy = {s: {a: 1.0 / 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:
            return policy, V
        policy = new_policy

if __name__ == "__main__":
    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:
[0.0, -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, 0.0]

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:

A **Convolutional Neural Network (CNN)** is a type of deep learning model specifically designed for processing data that has a grid-like topology, such as images. CNNs are highly effective in computer vision tasks because they automatically learn spatial hierarchies of features through convolutional layers. One of the most common datasets used to introduce CNNs is the **MNIST dataset**, which contains 70,000 grayscale images of handwritten digits (0–9), each of size 28×28 pixels.

Structure of CNN:
A typical CNN architecture consists of several types of layers:

1. **Convolutional Layer:** Applies a set of filters (kernels) that slide over the input image to extract local features such as edges, corners, and textures. Each filter produces a feature map that highlights a particular pattern in the input.

2. **Activation Function (ReLU):** Applies a non-linear transformation, typically the Rectified Linear Unit (ReLU), which replaces negative values with zero. This helps the network learn complex patterns and improves training efficiency.

3. **Pooling Layer:** Reduces the spatial dimensions of feature maps while preserving important information. Common types include max pooling and average pooling. Pooling helps make the model invariant to small translations and reduces computation.

4. **Fully Connected Layer (Dense):** After several convolutional and pooling layers, the output is flattened into a vector and fed into fully connected layers. These layers combine features to classify the input image into one of the ten digit categories.

5. **Output Layer:** The final layer uses the **Softmax** activation function to output probabilities corresponding to each digit class (0–9).

Training Process:
The training process involves feeding input images through the CNN and comparing its output predictions with the true labels using a **loss function**, typically categorical cross-entropy. The network's parameters (weights and biases) are optimized using **backpropagation** and an optimizer like **Stochastic Gradient Descent (SGD)** or **Adam**. During training, the CNN learns which features are most useful for distinguishing between digits.

Implementation on MNIST:
Using frameworks like TensorFlow or PyTorch, we can build and train a CNN on MNIST as follows:

- **Input:** 28×28 grayscale images normalized between 0 and 1.

- **Architecture Example:**

    - Convolutional Layer 1: 32 filters, 3×3 kernel, ReLU activation.
    - Pooling Layer 1: Max Pooling (2×2).
    - Convolutional Layer 2: 64 filters, 3×3 kernel, ReLU activation.
    - Pooling Layer 2: Max Pooling (2×2).
    - Fully Connected Layer: 128 neurons, ReLU activation.
    - Output Layer: 10 neurons, Softmax activation.
- **Training:** Typically done for 5–10 epochs with a batch size of 32 or 64.

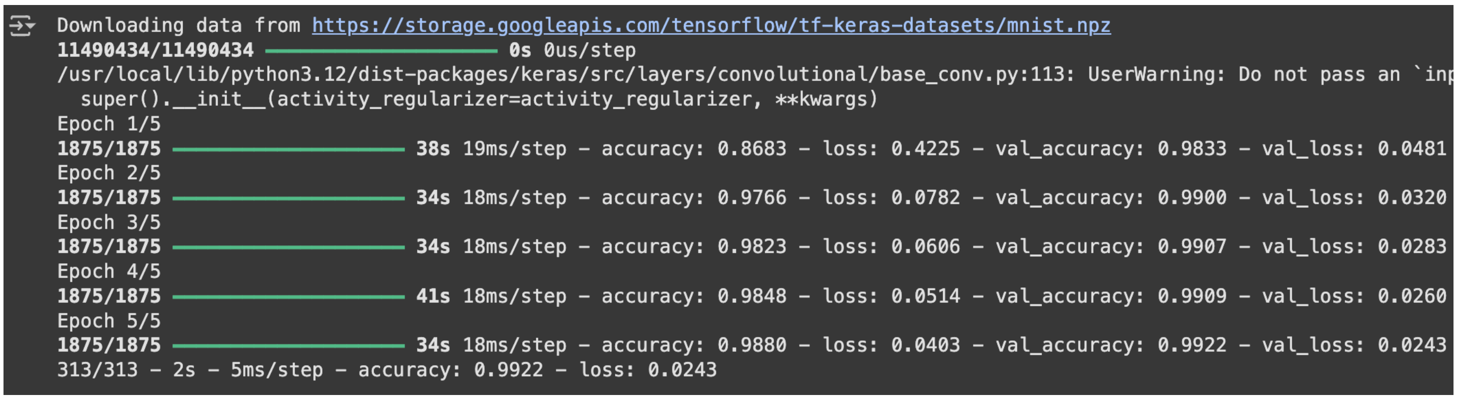
Evaluation:
After training, the CNN is evaluated on the test dataset (10,000 images). A well-trained CNN on MNIST typically achieves **accuracy above 99%**, demonstrating its strong ability to generalize across handwritten digits.

Conclusion:
CNNs revolutionized image recognition by eliminating the need for manual feature extraction. By automatically learning visual patterns at multiple levels of abstraction, they achieve remarkable accuracy on tasks like handwritten digit recognition. The MNIST dataset serves as an ideal starting point for understanding CNNs, offering a simple yet powerful demonstration of deep learning's effectiveness in image classification.

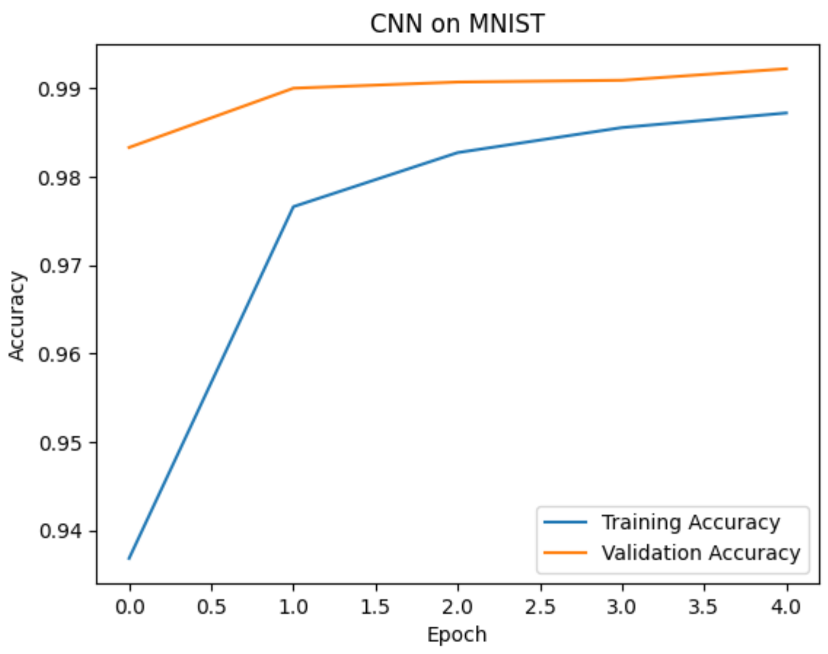
CODE:

```
import os
os.environ.setdefault("TF_CPP_MIN_LOG_LEVEL", "2")

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

# Load and preprocess MNIST
def load_data():
    (x_train, y_train), (x_test, y_test) = datasets.mnist.load_data()
    x_train = (x_train / 255.0).reshape((-1, 28, 28, 1))
    x_test = (x_test / 255.0).reshape((-1, 28, 28, 1))
    return (x_train, y_train), (x_test, y_test)

# Define model
def build_model():
    model = models.Sequential([
        layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28,
28, 1)),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Flatten(),
        layers.Dense(128, activation='relu'),
        layers.Dropout(0.5),
        layers.Dense(10, activation='softmax')
    ])
    model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model

if __name__ == "__main__":
    (x_train, y_train), (x_test, y_test) = load_data()
    model = build_model()
    history = model.fit(x_train, y_train, epochs=5,
validation_data=(x_test, y_test))
    test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
    print(f"\nTest Accuracy: {test_acc:.4f}")

    # Plot accuracy
    plt.plot(history.history['accuracy'], label='Training Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation
Accuracy')
    plt.title('CNN on MNIST')
    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 `inp
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 38s 19ms/step - accuracy: 0.8683 - loss: 0.4225 - val_accuracy: 0.9833 - val_loss: 0.0481
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 34s 18ms/step - accuracy: 0.9766 - loss: 0.0782 - val_accuracy: 0.9900 - val_loss: 0.0320
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 34s 18ms/step - accuracy: 0.9823 - loss: 0.0606 - val_accuracy: 0.9907 - val_loss: 0.0283
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 41s 18ms/step - accuracy: 0.9848 - loss: 0.0514 - val_accuracy: 0.9909 - val_loss: 0.0260
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 34s 18ms/step - accuracy: 0.9880 - loss: 0.0403 - val_accuracy: 0.9922 - val_loss: 0.0243
313/313 - 2s - 5ms/step - accuracy: 0.9922 - loss: 0.0243
```



LEARNING OUTCOME:

EXPERIMENT - 6

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

THEORY:

Sentiment Analysis is a natural language processing (NLP) technique used to determine the emotional tone of text — such as positive, negative, or neutral. A common benchmark dataset for this task is the **IMDB movie reviews dataset**, which contains 50,000 reviews labeled as either positive or negative. To effectively capture the sequential nature of text, we use a **Long Short-Term Memory (LSTM)** network, a special type of **Recurrent Neural Network (RNN)** designed to learn long-term dependencies in data.

Understanding LSTM:
Traditional RNNs struggle with long-term dependencies due to the **vanishing and exploding gradient problem** during training. LSTMs overcome this limitation by introducing a **memory cell** and **gating mechanisms** that control the flow of information. The three main gates in an LSTM are:

1. **Forget Gate:** Decides which information from the previous cell state should be discarded.
2. **Input Gate:** Determines which new information should be stored in the cell state.
3. **Output Gate:** Controls which information from the current cell state should be output.

This gating mechanism allows LSTMs to retain important context from earlier words in a sentence, making them ideal for tasks like sentiment analysis where word order and context matter.

Preprocessing the IMDB Dataset:
Before training, the text data must be preprocessed:

- **Tokenization:** Converting text into sequences of integers, where each integer represents a word.
- **Padding:** Ensuring all sequences have the same length by adding zeros to shorter ones.
- **Vocabulary Limiting:** Restricting the vocabulary to the most frequent words (e.g., top 10,000 words).

This preprocessing helps standardize the data for input into the neural network.

LSTM Model Architecture:
A typical LSTM model for IMDB sentiment analysis consists of the following layers:

1. **Embedding Layer:** Converts integer-encoded words into dense vector representations, capturing semantic meaning.
2. **LSTM Layer:** Processes the word embeddings sequentially, learning contextual relationships between words.
3. **Dense (Fully Connected) Layer:** Uses the learned features to make predictions.

4. **Output Layer:** A single neuron with a **sigmoid activation function** to classify reviews as positive (1) or negative (0).

An example architecture:

- Embedding layer with input_dim = 10,000 and output_dim = 128
- LSTM layer with 128 units
- Dense layer with 1 unit and sigmoid activation

Training Process:
The model is compiled with **binary cross-entropy loss** and optimized using **Adam optimizer**. Training is typically done over multiple epochs (e.g., 5–10) with batches of 64 samples each. During training, the model learns to associate patterns in word sequences with sentiment polarity.

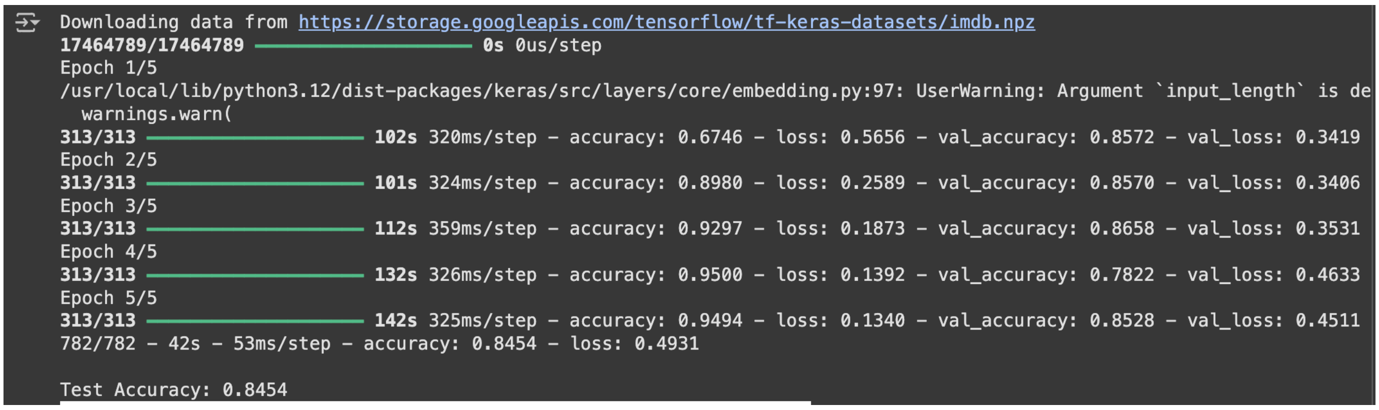
Evaluation:
Once trained, the model is evaluated on a test dataset. LSTM models generally achieve high accuracy (around **85–90%**) on IMDB sentiment classification tasks. The model can then be used to predict sentiment for new or unseen reviews by preprocessing the text and feeding it into the trained network.

Conclusion:
Using **LSTM networks** for **IMDB sentiment analysis** effectively captures the sequential and contextual nature of human language. Unlike traditional machine learning models that rely on handcrafted features, LSTMs learn complex linguistic patterns directly from data, making them a powerful tool for text-based sentiment prediction. This application demonstrates the strength of deep learning in understanding emotions and opinions expressed in natural language.

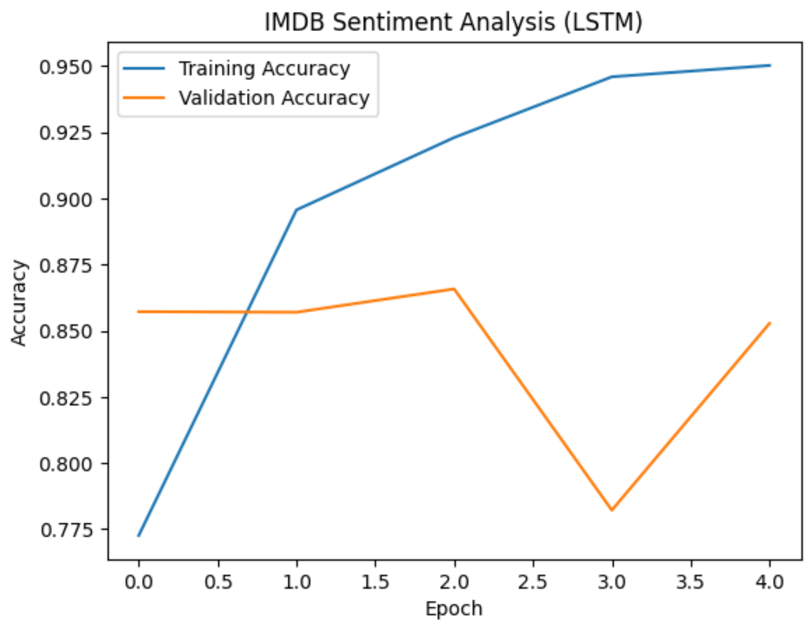
CODE:

```
import os
os.environ.setdefault("TF_CPP_MIN_LOG_LEVEL", "2")

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

num_words = 10000
maxlen = 200

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

x_train = preprocessing.sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = preprocessing.sequence.pad_sequences(x_test, maxlen=maxlen)

model = models.Sequential([
    layers.Embedding(input_dim=num_words, output_dim=128,
input_length=maxlen),
    layers.LSTM(128),
    layers.Dropout(0.5),
    layers.Dense(64, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

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

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

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 ━━━━━━━━━━━━━━━━━━━━ 102s 320ms/step - accuracy: 0.6746 - loss: 0.5656 - val_accuracy: 0.8572 - val_loss: 0.3419
Epoch 2/5
313/313 ━━━━━━━━━━━━━━━━━━━━ 101s 324ms/step - accuracy: 0.8980 - loss: 0.2589 - val_accuracy: 0.8570 - val_loss: 0.3406
Epoch 3/5
313/313 ━━━━━━━━━━━━━━━━━━━━ 112s 359ms/step - accuracy: 0.9297 - loss: 0.1873 - val_accuracy: 0.8658 - val_loss: 0.3531
Epoch 4/5
313/313 ━━━━━━━━━━━━━━━━━━━━ 132s 326ms/step - accuracy: 0.9500 - loss: 0.1392 - val_accuracy: 0.7822 - val_loss: 0.4633
Epoch 5/5
313/313 ━━━━━━━━━━━━━━━━━━━━ 142s 325ms/step - accuracy: 0.9494 - loss: 0.1340 - val_accuracy: 0.8528 - val_loss: 0.4511
782/782 - 42s - 53ms/step - accuracy: 0.8454 - loss: 0.4931

Test Accuracy: 0.8454
```



LEARNING OUTCOME:

EXPERIMENT - 7

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

THEORY:

Transfer Learning is a deep learning technique where a model trained on a large dataset is reused or fine-tuned for a different but related task. This approach significantly reduces training time and improves performance, especially when the new dataset is smaller. One of the most popular pretrained models used in image classification is **VGG16**, a deep convolutional neural network developed by the Visual Geometry Group at Oxford. It was trained on the **ImageNet dataset**, which contains over a million images across 1,000 categories.

The **CIFAR-10 dataset** is a smaller benchmark dataset consisting of 60,000 color images (32×32 pixels) categorized into 10 classes such as airplanes, cars, birds, and cats. Since VGG16 expects input images of size **224×224**, the CIFAR-10 images are resized accordingly before being passed through the network.

VGG16 Architecture Overview:
VGG16 is composed of 16 layers with learnable parameters—13 convolutional layers and 3 fully connected layers. It uses small 3×3 convolution filters and 2×2 max-pooling layers, allowing it to capture fine-grained features from images. The final dense layers perform classification based on the extracted features.

Transfer Learning Process:
In this setup, the convolutional base of VGG16 (trained on ImageNet) is reused, as it already contains generalized feature detectors such as edge and shape recognizers. The final classification layers are replaced and retrained on the CIFAR-10 dataset to adapt the model to the new categories. The steps involved are:

1. **Load Pretrained Model:**
   Import the VGG16 model from Keras with `weights='imagenet'`, excluding the top (fully connected) layers by setting `include_top=False`.
2. **Freeze Convolutional Layers:**
   Freeze the weights of the pretrained convolutional base so that only the new top layers are trained. This prevents loss of the general features learned from ImageNet.
3. **Add Custom Layers:**
   Add new layers such as:
   - Global Average Pooling or Flatten layer
   - Fully Connected (Dense) layer with ReLU activation
   - Dropout layer for regularization
   - Output layer with 10 neurons and Softmax activation (for the 10 CIFAR-10 classes)

4. **Preprocess and Resize Images:**
   CIFAR-10 images are resized from 32×32×3 to **224×224×3** to match the VGG16 input size using `tf.image.resize`.

5. **Compile and Train:**
   Compile the model using the **Adam optimizer** and **categorical cross-entropy loss**. The training process fine-tunes the new top layers on the CIFAR-10 dataset, while the pretrained base retains its learned feature representations.

Benefits of Transfer Learning:

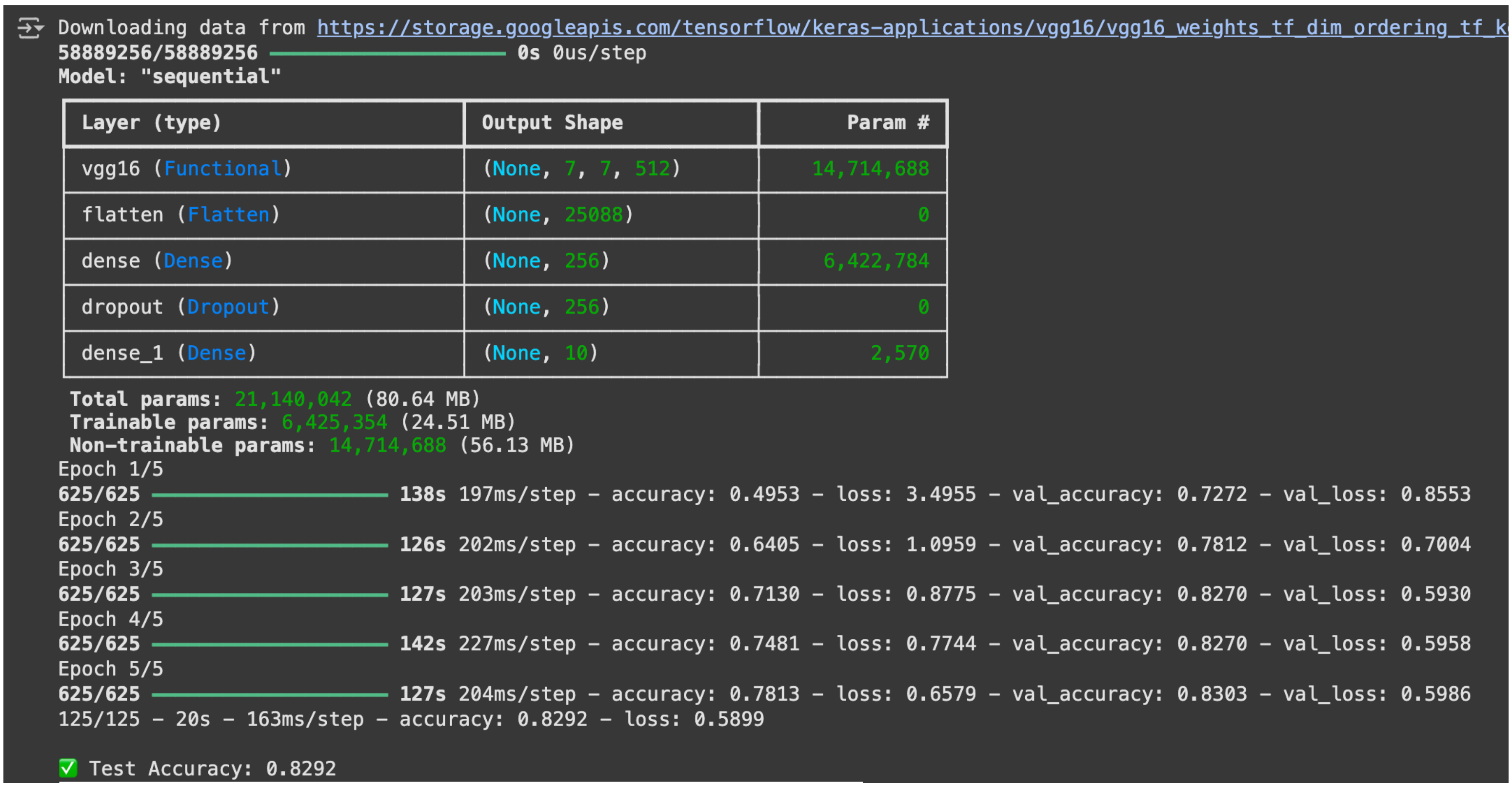
- Reduces computational cost and training time.
- Prevents overfitting, especially when the dataset is small.
- Utilizes robust and general features learned from large-scale datasets.

Evaluation:
After training, the model is evaluated on the test set. Typically, transfer learning with VGG16 on CIFAR-10 achieves an accuracy between **85–90%**, outperforming models trained from scratch on the same data.

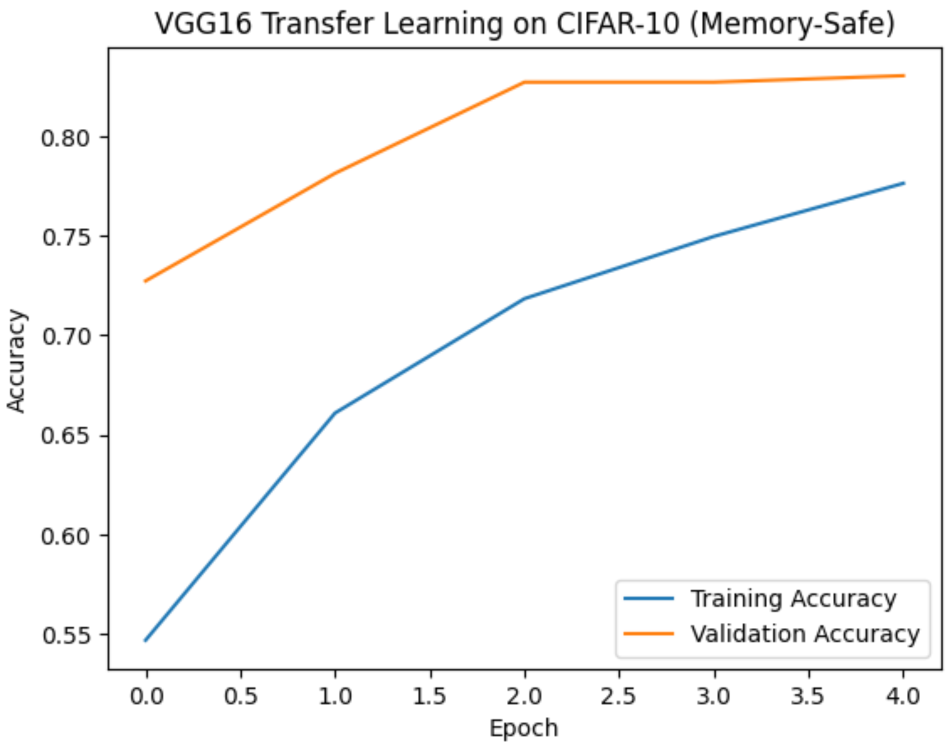
CODE:

```
import os
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "2"

import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.applications import VGG16
from tensorflow.keras.applications.vgg16 import preprocess_input
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt


#  Load CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# (Optional) Reduce dataset size to fit memory limits comfortably
x_train = x_train[:20000]
y_train = y_train[:20000]
x_test = x_test[:4000]
y_test = y_test[:4000]


#  Create ImageDataGenerators
train_datagen = ImageDataGenerator(
    preprocessing_function=preprocess_input
)
test_datagen = ImageDataGenerator(
    preprocessing_function=preprocess_input
)

# Batch generator with on-the-fly resizing to 224x224
def generator(datagen, x, y, batch_size=32):
    for batch_x, batch_y in datagen.flow(x, y, batch_size=batch_size):
        batch_x_resized = tf.image.resize(batch_x, (224, 224))
        yield batch_x_resized, batch_y

batch_size = 32
train_steps = len(x_train) // batch_size
test_steps = len(x_test) // batch_size

train_gen = generator(train_datagen, x_train, y_train, batch_size)
test_gen = generator(test_datagen, x_test, y_test, batch_size)


# Define Transfer Learning Model
base_model = VGG16(weights='imagenet', include_top=False,
input_shape=(224, 224, 3))
for layer in base_model.layers:
    layer.trainable = False

model = models.Sequential([
    base_model,
    layers.Flatten(),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(10, activation='softmax')
])

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

model.summary()


# 4 Train the Model
history = model.fit(
    train_gen,
    steps_per_epoch=train_steps,
    validation_data=test_gen,
    validation_steps=test_steps,
    epochs=5
)

# 5 Evaluate and Plot
test_loss, test_acc = model.evaluate(test_gen, steps=test_steps,
verbose=2)
print(f"\n✅ Test Accuracy: {test_acc:.4f}")

plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('VGG16 Transfer Learning on CIFAR-10 (Memory-Safe)')
plt.xlabel('Epoch'); plt.ylabel('Accuracy'); plt.legend(); plt.show()
```

OUTPUT:



```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_k
58889256/58889256 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Model: "sequential"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ vgg16 (Functional)              │ (None, 7, 7, 512)      │    14,714,688 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ flatten (Flatten)               │ (None, 25088)          │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense (Dense)                   │ (None, 256)            │     6,422,784 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout (Dropout)               │ (None, 256)            │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_1 (Dense)                 │ (None, 10)             │         2,570 │
└─────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 21,140,042 (80.64 MB)
 Trainable params: 6,425,354 (24.51 MB)
 Non-trainable params: 14,714,688 (56.13 MB)
Epoch 1/5
625/625 ━━━━━━━━━━━━━━━━━━━━ 138s 197ms/step - accuracy: 0.4953 - loss: 3.4955 - val_accuracy: 0.7272 - val_loss: 0.8553
Epoch 2/5
625/625 ━━━━━━━━━━━━━━━━━━━━ 126s 202ms/step - accuracy: 0.6405 - loss: 1.0959 - val_accuracy: 0.7812 - val_loss: 0.7004
Epoch 3/5
625/625 ━━━━━━━━━━━━━━━━━━━━ 127s 203ms/step - accuracy: 0.7130 - loss: 0.8775 - val_accuracy: 0.8270 - val_loss: 0.5930
Epoch 4/5
625/625 ━━━━━━━━━━━━━━━━━━━━ 142s 227ms/step - accuracy: 0.7481 - loss: 0.7744 - val_accuracy: 0.8270 - val_loss: 0.5958
Epoch 5/5
625/625 ━━━━━━━━━━━━━━━━━━━━ 127s 204ms/step - accuracy: 0.7813 - loss: 0.6579 - val_accuracy: 0.8303 - val_loss: 0.5986
125/125 - 20s - 163ms/step - accuracy: 0.8292 - loss: 0.5899

✅ Test Accuracy: 0.8292
```



LEARNING OUTCOME:

EXPERIMENT - 8

AIM: Image Caption Generator using CNN + LSTM

THEORY:

Image Captioning is an advanced deep learning task that combines **computer vision** and **natural language processing (NLP)** to generate descriptive sentences for images. The goal is to enable machines to "see" and "describe" visual content in human-like language. A common and effective architecture for image captioning integrates **Convolutional Neural Networks (CNNs)** for feature extraction from images and **Long Short-Term Memory (LSTM)** networks for generating the textual sequence (caption).

In this toy demonstration, we use **InceptionV3**, a powerful CNN pretrained on the ImageNet dataset, as the **feature extractor**, and pair it with an **LSTM** network to simulate the caption generation process using random toy sequences.

Architecture Overview:
The image captioning pipeline consists of two main components:

1. **CNN Encoder (Feature Extractor):**
   - A pretrained **InceptionV3** network (from Keras applications) is used to extract high-level visual features from an image.
   - The final classification layers are removed (`include_top=False`), and the output of the last convolutional block is used as the feature vector.
   - This feature vector represents the semantic content of the image, such as objects, colors, and spatial arrangements.
2. **LSTM Decoder (Sequence Generator):**
   - The extracted image features are fed into an **LSTM network** that generates words sequentially to form a caption.
   - The LSTM learns temporal dependencies in the sequence, predicting each next word based on the image context and previously generated words.
   - In this toy demo, we use random toy sequences (e.g., numeric or symbolic sequences) instead of real captions to demonstrate how the CNN and LSTM components are connected.

Model Wiring Process:

1. **Feature Extraction:**
   - An image is passed through InceptionV3, resized to 299×299 pixels, and preprocessed using the `preprocess_input` function.
   - The resulting feature vector is flattened or passed through a dense layer to reduce dimensionality (e.g., 2048 → 256).

2. **Sequence Input:**
   - A sequence of tokens (or random toy inputs) is embedded using an **Embedding layer**, converting each token into a dense vector representation.
3. **Merging Features:**
   - The CNN feature vector and the LSTM input sequence are combined, typically by concatenating or adding them before passing to the decoder.
   - The merged output is processed by one or more LSTM layers, followed by a Dense layer with **Softmax activation** to predict the next word in the sequence.
4. **Training (Toy Example):**
   - In a real setup, the model is trained on paired image–caption datasets (like MS COCO).
   - In this toy demo, random data is used to show model connectivity and flow without training on real captions.

Key Advantages of CNN + LSTM Architecture:

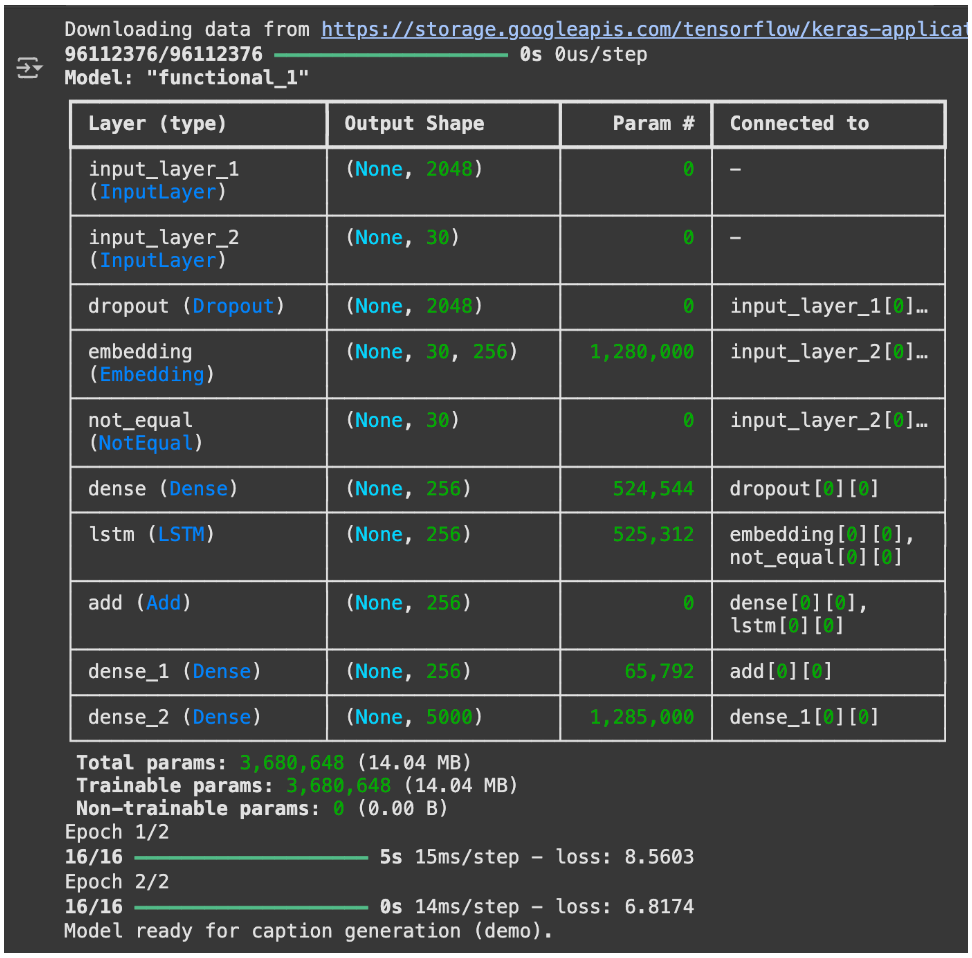
- **Feature Learning:** CNNs automatically extract rich and meaningful image representations.
- **Sequence Modeling:** LSTMs capture word dependencies, ensuring grammatically coherent captions.
- **End-to-End Learning:** The system can be trained jointly to minimize caption prediction loss across images.

CODE:

```
import os
os.environ.setdefault("TF_CPP_MIN_LOG_LEVEL", "2")

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 feature extractor
cnn_model = InceptionV3(weights='imagenet')
cnn_model = models.Model(cnn_model.input, cnn_model.layers[-2].output)

# 2. Toy data
vocab_size = 5000
max_length = 30
X1 = np.random.randn(512, 2048).astype('float32')
X2 = np.random.randint(1, vocab_size, (512, max_length))
```

```
y = np.random.randint(1, vocab_size, (512, 1))

# 3. Model definition
inputs1 = Input(shape=(2048,))
fe1 = layers.Dropout(0.5)(inputs1)
fe2 = layers.Dense(256, activation='relu')(fe1)

inputs2 = Input(shape=(max_length,))
se1 = layers.Embedding(vocab_size, 256, mask_zero=True)(inputs2)
se2 = layers.LSTM(256)(se1)

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()

y_onehot = to_categorical(y, num_classes=vocab_size)
history = model.fit([X1, X2], y_onehot, epochs=2, batch_size=32,
verbose=1)

print("Model ready for caption generation (demo).")
```

OUTPUT:



```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applicat
96112376/96112376 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Model: "functional_1"
┏━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━┓
┃ Layer (type)         ┃ Output Shape     ┃    Param # ┃ Connected to      ┃
┡━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━┩
│ input_layer_1        │ (None, 2048)     │          0 │ -                 │
│ (InputLayer)         │                  │            │                   │
├──────────────────────┼──────────────────┼────────────┼───────────────────┤
│ input_layer_2        │ (None, 30)       │          0 │ -                 │
│ (InputLayer)         │                  │            │                   │
├──────────────────────┼──────────────────┼────────────┼───────────────────┤
│ dropout (Dropout)    │ (None, 2048)     │          0 │ input_layer_1[0]… │
├──────────────────────┼──────────────────┼────────────┼───────────────────┤
│ embedding            │ (None, 30, 256)  │  1,280,000 │ input_layer_2[0]… │
│ (Embedding)          │                  │            │                   │
├──────────────────────┼──────────────────┼────────────┼───────────────────┤
│ not_equal            │ (None, 30)       │          0 │ input_layer_2[0]… │
│ (NotEqual)           │                  │            │                   │
├──────────────────────┼──────────────────┼────────────┼───────────────────┤
│ dense (Dense)        │ (None, 256)      │    524,544 │ dropout[0][0]     │
├──────────────────────┼──────────────────┼────────────┼───────────────────┤
│ lstm (LSTM)          │ (None, 256)      │    525,312 │ embedding[0][0],  │
│                      │                  │            │ not_equal[0][0]   │
├──────────────────────┼──────────────────┼────────────┼───────────────────┤
│ add (Add)            │ (None, 256)      │          0 │ dense[0][0],      │
│                      │                  │            │ lstm[0][0]        │
├──────────────────────┼──────────────────┼────────────┼───────────────────┤
│ dense_1 (Dense)      │ (None, 256)      │     65,792 │ add[0][0]         │
├──────────────────────┼──────────────────┼────────────┼───────────────────┤
│ dense_2 (Dense)      │ (None, 5000)     │  1,285,000 │ dense_1[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 1/2
16/16 ━━━━━━━━━━━━━━━━━━━━ 5s 15ms/step - loss: 8.5603
Epoch 2/2
16/16 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - loss: 6.8174
Model ready for caption generation (demo).
```

LEARNING OUTCOME:

EXPERIMENT - 9

AIM: Face Mask Detection using CNN

THEORY:

Face Mask Detection is a computer vision application that automatically determines whether a person in an image is wearing a mask or not. This technology became especially significant during the COVID-19 pandemic, helping ensure safety and compliance in public spaces. The task is essentially a **binary image classification problem**, where the two classes are *"Mask"* and *"No Mask."* A **Convolutional Neural Network (CNN)** is used for this task due to its strong ability to learn visual features directly from image data.

Dataset and Directory Structure:
The dataset for face mask detection is typically organized in a **directory-based structure**, which is compatible with libraries like TensorFlow and Keras. The directory structure might look like this:

```
dataset/
├── train/
│   ├── with_mask/
│   └── without_mask/
├── val/
│   ├── with_mask/
│   └── without_mask/
└── test/
    ├── with_mask/
    └── without_mask/
```

Each subfolder contains corresponding labeled images. The dataset loader automatically infers class labels based on folder names, simplifying data handling.

Data Loading and Preprocessing:
Using the **ImageDataGenerator** or **tf.keras.utils.image_dataset_from_directory**, images are:

- Loaded from directories and resized (commonly to 128×128 or 224×224 pixels).
- Rescaled to normalize pixel values between 0 and 1.
- Augmented using transformations like rotation, flipping, and zooming to improve model generalization.

This preprocessing helps the CNN handle variations in lighting, pose, and orientation.

CNN Architecture:
A typical CNN for mask detection includes several convolutional and pooling layers followed by dense layers. Example architecture:

1. **Convolutional Layers:**

    - Use multiple filters (e.g., 32, 64, 128) with 3×3 kernels to detect features such as edges and shapes.
    - Each convolutional layer is followed by a **ReLU activation** to introduce non-linearity.

2. **Pooling Layers:**
    - **Max Pooling** (2×2) reduces spatial dimensions and computational cost while retaining important information.
3. **Dropout Layer:**
    - Randomly deactivates neurons during training to prevent overfitting.
4. **Flatten Layer:**
    - Converts the 2D feature maps into a 1D vector for classification.
5. **Fully Connected (Dense) Layers:**
    - Combine extracted features to classify the input image.
    - The final Dense layer uses a **sigmoid activation** (for binary classification).

Model Compilation and Training:
The model is compiled using:

- **Loss Function:** Binary Cross-Entropy
- **Optimizer:** Adam or RMSprop
- **Metrics:** Accuracy

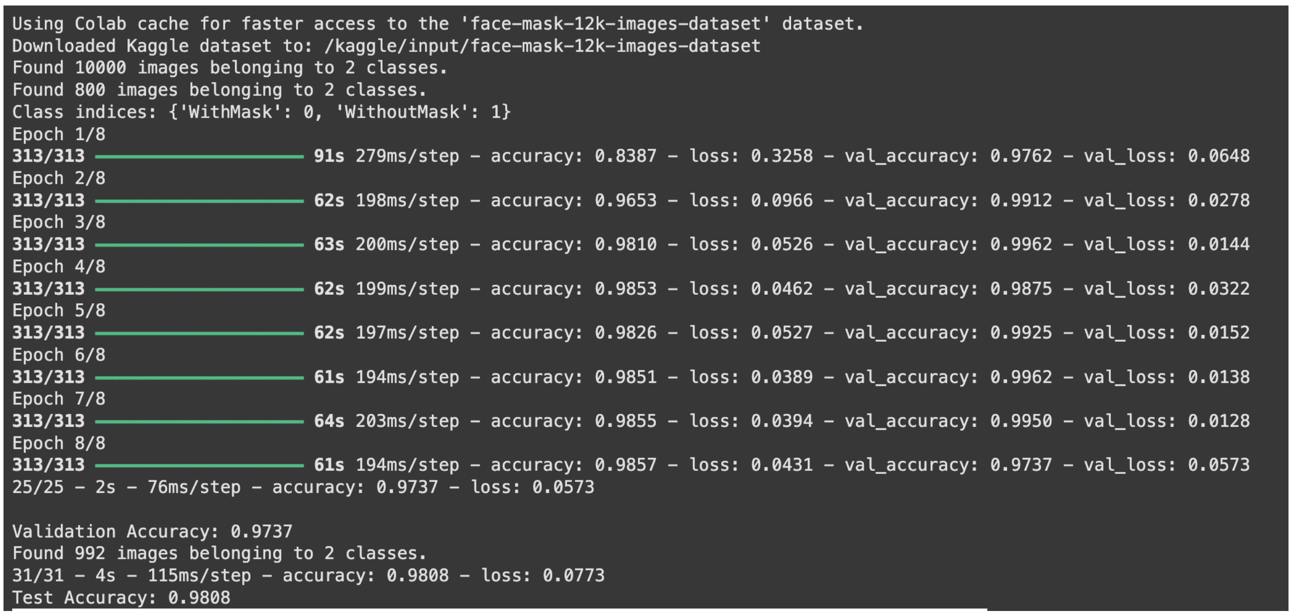
The model is trained for several epochs, typically between 10–30, with validation data to monitor performance and avoid overfitting.

Evaluation and Prediction:
After training, the model is evaluated on the test set. The trained CNN achieves high accuracy (typically above **95%**) on well-labeled datasets. The model can then be used to predict new images or integrated into real-time systems using webcam input.

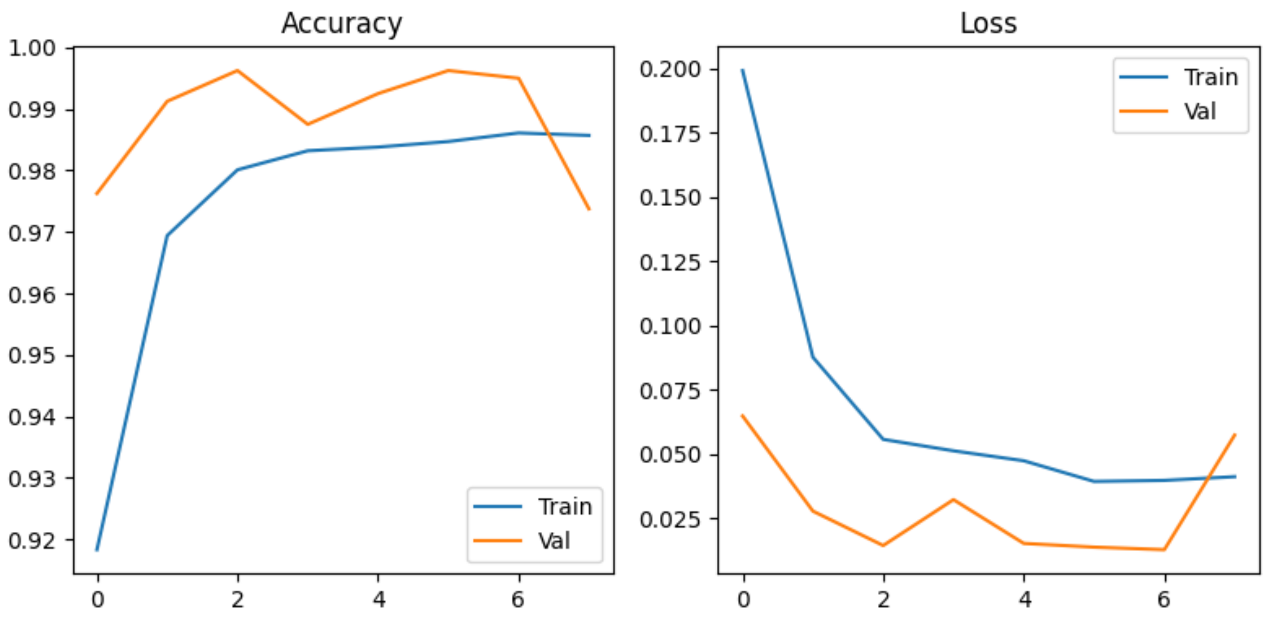
CODE:

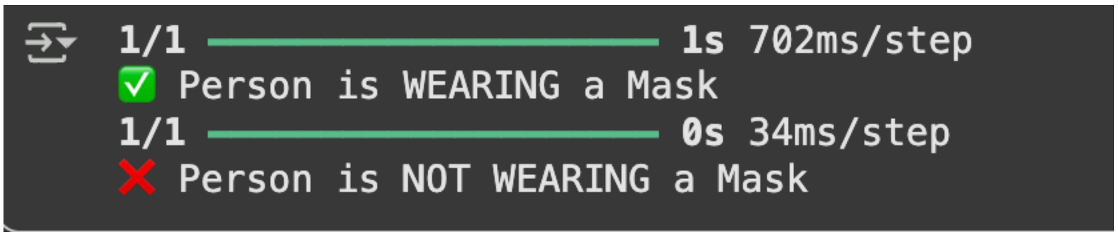
```
import os
os.environ.setdefault("TF_CPP_MIN_LOG_LEVEL", "2")

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

# Try to fetch dataset via kagglehub (public dataset)
DATASET_COORD = "ashishjangra27/face-mask-12k-images-dataset"

def try_download_via_kagglehub():
    try:
        import kagglehub  # type: ignore
    except Exception as e:
        print("kagglehub not available:", e)
        return None
    try:
        path = kagglehub.dataset_download(DATASET_COORD)
        print("Downloaded Kaggle dataset to:", path)
        return path
    except Exception as e:
        print("kagglehub download failed:", e)
        return None

def find_split_dirs(root_path: str):
    """Find Train/Test/Validation directories inside root_path, case-
insensitive.
    Returns (train_dir, val_dir, test_dir) where any may be None if not
found.
    """
    # Search common names and casing
    candidates = []
    for dirpath, dirnames, _ in os.walk(root_path):
        base = os.path.basename(dirpath).lower()
        if base in {"train", "training"}:
            candidates.append(("train", dirpath))
        elif base in {"val", "valid", "validation"}:
            candidates.append(("val", dirpath))
        elif base in {"test", "testing"}:
            candidates.append(("test", dirpath))
        # stop descending too deep to keep search fast
        if dirpath.count(os.sep) - root_path.count(os.sep) > 3:
            break
    train_dir = next((p for k, p in candidates if k == "train"), None)
    val_dir = next((p for k, p in candidates if k == "val"), None)
    test_dir = next((p for k, p in candidates if k == "test"), None)
    return train_dir, val_dir, test_dir

# Resolve dataset root
dataset_root = try_download_via_kagglehub()
if not dataset_root:
    dataset_root = os.environ.get("FACE_MASK_DATASET_ROOT", "data")
    print("Using local dataset root:", dataset_root)

# Some Kaggle datasets nest files in a child folder like "Face Mask
Dataset"
# If so, prefer that as the root for finding splits
preferred_child = None
if os.path.isdir(dataset_root):
    # look for a child folder containing both 'with' and 'without'
classes or split dirs
    for name in os.listdir(dataset_root):
        p = os.path.join(dataset_root, name)
        if os.path.isdir(p) and any(x.lower() in name.lower() for x in
["face mask", "facemask", "dataset"]):
            preferred_child = p
            break
if preferred_child:
    dataset_root = preferred_child

train_dir, val_dir, test_dir = find_split_dirs(dataset_root)
if not train_dir or not val_dir:
    raise SystemExit(
        f"Could not locate Train/Validation folders under
{dataset_root}.\n"
        "Make sure the dataset is extracted and available, or set
FACE_MASK_DATASET_ROOT."
    )

# Image parameters
img_size = (128, 128)
batch_size = 32

# Data generators without split (directories already split)
train_datagen = ImageDataGenerator(
    rescale=1.0 / 255.0,
    rotation_range=20,
    zoom_range=0.2,
    shear_range=0.2,
    horizontal_flip=True,
)

val_datagen = ImageDataGenerator(rescale=1.0 / 255.0)

train_data = train_datagen.flow_from_directory(
    train_dir,
    target_size=img_size,
    batch_size=batch_size,
    class_mode='binary',
    shuffle=True,
)

val_data = val_datagen.flow_from_directory(
    val_dir,
    target_size=img_size,
    batch_size=batch_size,
    class_mode='binary',
    shuffle=False,
)
print("Class indices:", train_data.class_indices)

# Model
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu',
input_shape=(img_size[0], img_size[1], 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
])

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

# Training
history = model.fit(
    train_data,
    epochs=8,
    validation_data=val_data,
)

# Evaluation on validation and optional test split
val_loss, val_acc = model.evaluate(val_data, verbose=2)
print(f"\nValidation Accuracy: {val_acc:.4f}")
if test_dir:
    test_datagen = ImageDataGenerator(rescale=1.0 / 255.0)
    test_data = test_datagen.flow_from_directory(
        test_dir,
        target_size=img_size,
        batch_size=batch_size,
        class_mode='binary',
        shuffle=False,
    )
    test_loss, test_acc = model.evaluate(test_data, verbose=2)
    print(f"Test Accuracy: {test_acc:.4f}")

# Plots
plt.figure(figsize=(8, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train')
plt.plot(history.history['val_accuracy'], label='Val')
plt.title('Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train')
plt.plot(history.history['val_loss'], label='Val')
plt.title('Loss')
plt.legend()
plt.tight_layout()
plt.show()

# Predict on Single Image (Example)
import numpy as np
from tensorflow.keras.preprocessing import 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]

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

predict_mask("/content/with_mask_12.jpg")
predict_mask("/content/without_mask_158.jpg")
```

OUTPUT:



```
Using Colab cache for faster access to the 'face-mask-12k-images-dataset' dataset.
Downloaded Kaggle dataset to: /kaggle/input/face-mask-12k-images-dataset
Found 10000 images belonging to 2 classes.
Found 800 images belonging to 2 classes.
Class indices: {'WithMask': 0, 'WithoutMask': 1}
Epoch 1/8
313/313 ━━━━━━━━━━━━━━━━━━━━ 91s 279ms/step - accuracy: 0.8387 - loss: 0.3258 - val_accuracy: 0.9762 - val_loss: 0.0648
Epoch 2/8
313/313 ━━━━━━━━━━━━━━━━━━━━ 62s 198ms/step - accuracy: 0.9653 - loss: 0.0966 - val_accuracy: 0.9912 - val_loss: 0.0278
Epoch 3/8
313/313 ━━━━━━━━━━━━━━━━━━━━ 63s 200ms/step - accuracy: 0.9810 - loss: 0.0526 - val_accuracy: 0.9962 - val_loss: 0.0144
Epoch 4/8
313/313 ━━━━━━━━━━━━━━━━━━━━ 62s 199ms/step - accuracy: 0.9853 - loss: 0.0462 - val_accuracy: 0.9875 - val_loss: 0.0322
Epoch 5/8
313/313 ━━━━━━━━━━━━━━━━━━━━ 62s 197ms/step - accuracy: 0.9826 - loss: 0.0527 - val_accuracy: 0.9925 - val_loss: 0.0152
Epoch 6/8
313/313 ━━━━━━━━━━━━━━━━━━━━ 61s 194ms/step - accuracy: 0.9851 - loss: 0.0389 - val_accuracy: 0.9962 - val_loss: 0.0138
Epoch 7/8
313/313 ━━━━━━━━━━━━━━━━━━━━ 64s 203ms/step - accuracy: 0.9855 - loss: 0.0394 - val_accuracy: 0.9950 - val_loss: 0.0128
Epoch 8/8
313/313 ━━━━━━━━━━━━━━━━━━━━ 61s 194ms/step - accuracy: 0.9857 - loss: 0.0431 - val_accuracy: 0.9737 - val_loss: 0.0573
25/25 - 2s - 76ms/step - accuracy: 0.9737 - loss: 0.0573

Validation Accuracy: 0.9737
Found 992 images belonging to 2 classes.
31/31 - 4s - 115ms/step - accuracy: 0.9808 - loss: 0.0773
Test Accuracy: 0.9808
```




```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 702ms/step
✅ Person is WEARING a Mask
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step
❌ Person is NOT WEARING a Mask
```

LEARNING OUTCOME:

EXPERIMENT - 10

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

THEORY:

Named Entity Recognition (NER) is a key task in **Natural Language Processing (NLP)** that involves identifying and classifying entities such as people, organizations, locations, dates, and more within a text. For example, in the sentence *"Barack Obama was born in Hawaii,"* the model should recognize *"Barack Obama"* as a **Person** and *"Hawaii"* as a **Location**. To achieve high accuracy in NER, models must capture both **contextual dependencies** and **sequential structure** in language. A highly effective architecture for this task combines **Bidirectional Long Short-Term Memory (Bi-LSTM)** networks with a **Conditional Random Field (CRF)** layer.

Bi-LSTM for Context Understanding:
A **Long Short-Term Memory (LSTM)** network is a type of **Recurrent Neural Network (RNN)** that can learn long-term dependencies in sequences. In **Bi-LSTM**, two LSTMs are used—one processes the sequence from left to right (forward direction), and the other processes it from right to left (backward direction). This bidirectional structure allows the model to incorporate both **past** and **future** context when predicting a tag for each word.

For instance, to identify the entity type of the word "Apple" in the sentence *"Apple released a new iPhone,"* the model needs to know the surrounding context — "released" indicates that "Apple" is likely an organization, not a fruit. The Bi-LSTM effectively captures this context.

Conditional Random Field (CRF) for Sequence Labeling:
While the Bi-LSTM provides contextual embeddings for each word, it treats each tag prediction independently. However, in NER tasks, the relationship between neighboring tags is crucial — for example, a tag *I-PER* (inside a person's name) cannot logically follow *B-LOC* (beginning of a location).

To handle this, a **CRF layer** is added on top of the Bi-LSTM output. The CRF models the conditional dependencies between output tags and ensures valid tag sequences by considering the entire sentence during prediction. This joint optimization improves sequence labeling consistency and overall accuracy.

Model Architecture:

1. **Input Layer:** Tokenized text is converted into sequences of word indices or embeddings (e.g., Word2Vec, GloVe, or contextual embeddings like BERT).

2. **Embedding Layer:** Converts tokens into dense vectors capturing semantic meaning.

3. **Bi-LSTM Layer:** Processes embeddings in both forward and backward directions to capture complete contextual information.

4. **CRF Layer:** Takes the Bi-LSTM outputs and predicts the most probable sequence of entity tags while enforcing tag consistency.

Training Process:
The model is trained using supervised learning with labeled sentences where each word has a

corresponding entity tag (e.g., *O*, *B-PER*, *I-ORG*). The **log-likelihood loss** function is used, and optimization is performed using algorithms like Adam. During training, the CRF layer learns transition probabilities between tags, while the Bi-LSTM learns contextual dependencies in the input text.

Advantages:

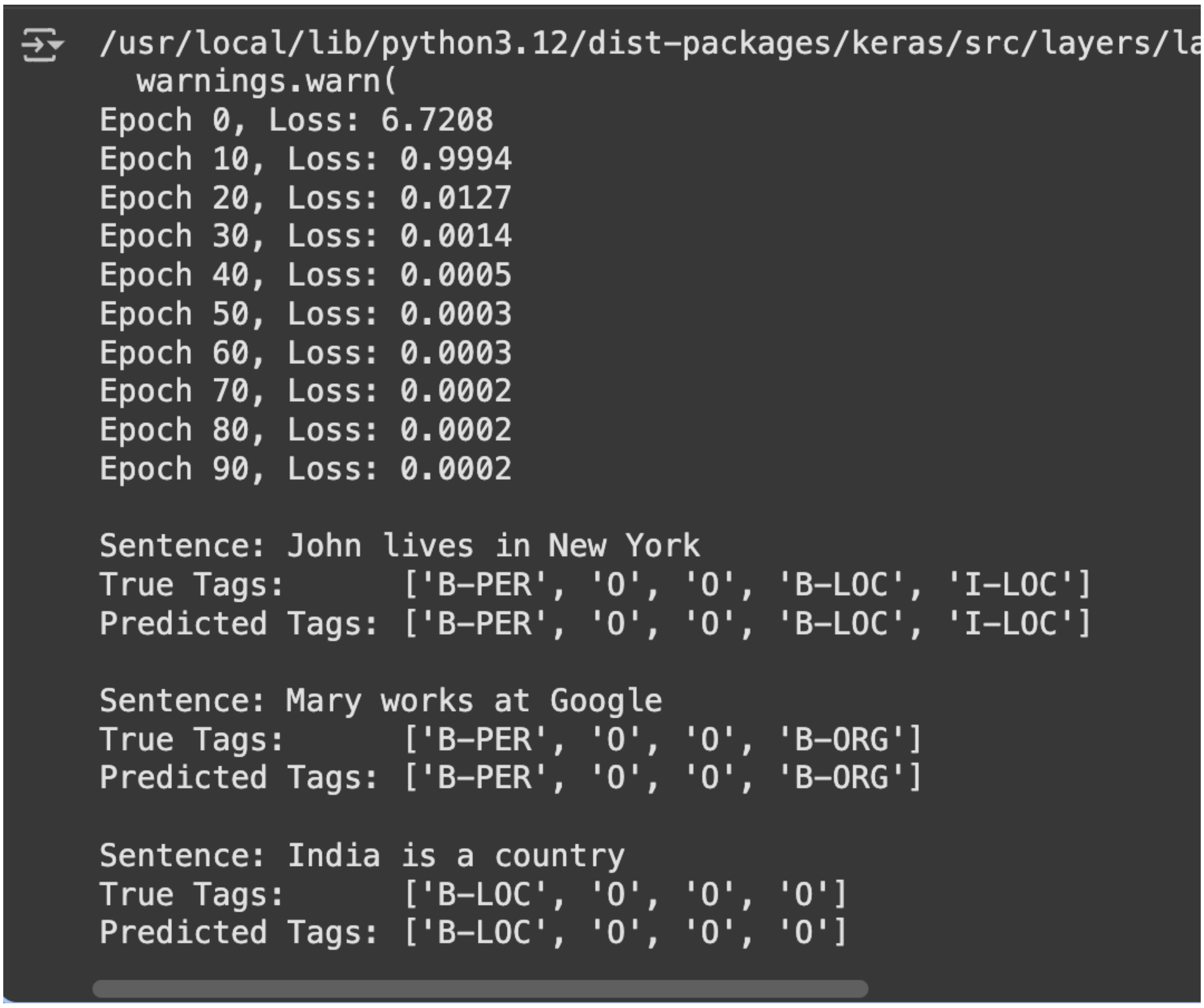
- **Context-aware:** Bi-LSTM captures information from both directions of a sentence.
- **Sequence-consistent:** CRF ensures valid and logical tag transitions.
- **High accuracy:** The Bi-LSTM + CRF combination significantly outperforms traditional NER models based on simple RNNs or feature-based approaches.

CODE:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, Model
from tensorflow.keras.preprocessing.sequence import pad_sequences

# ========== 1. Toy Data ==========
sentences = [
    ["John", "lives", "in", "New", "York"],
    ["Mary", "works", "at", "Google"],
    ["India", "is", "a", "country"],
]
tags = [
    ["B-PER", "O", "O", "B-LOC", "I-LOC"],
    ["B-PER", "O", "O", "B-ORG"],
    ["B-LOC", "O", "O", "O"],
]

# ========== 2. Preprocessing ==========
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)}
word2idx["PAD"], word2idx["OOV"] = 0, 1
tag2idx = {t: i for i, t in enumerate(tags_set)}
idx2tag = {i: t for t, i in tag2idx.items()}

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")

num_tags = len(tags_set)
vocab_size = len(word2idx)
mask = (X != 0)


# ========== 3. CRF Layer ==========
class CRFLayer(tf.keras.layers.Layer):
    def __init__(self, num_tags):
        super().__init__()
        self.num_tags = num_tags

    def build(self, input_shape):
        self.transitions = self.add_weight(
            shape=(self.num_tags, self.num_tags),
            initializer="glorot_uniform",
            trainable=True,
            name="transitions"
        )

    def call(self, logits, mask=None):
        seq_len = tf.shape(logits)[1]
        mask = tf.cast(mask, tf.float32)
        log_alpha = logits[:, 0]

        for t in range(1, seq_len):
            emit = logits[:, t]
            log_alpha_expanded = tf.expand_dims(log_alpha, 2)
            trans_expanded = tf.expand_dims(self.transitions, 0)
            scores = log_alpha_expanded + trans_expanded +
tf.expand_dims(emit, 1)
            log_alpha = tf.math.reduce_logsumexp(scores, axis=1) *
mask[:, t:t+1] + \
                        log_alpha * (1 - mask[:, t:t+1])

        log_Z = tf.math.reduce_logsumexp(log_alpha, axis=1)
        return log_Z

    def loss(self, logits, tags, mask):
        if not hasattr(self, "transitions"):
            self.build(tf.shape(logits))
        log_Z = self.call(logits, mask)
        mask = tf.cast(mask, tf.float32)
        batch_size = tf.shape(logits)[0]
        seq_len = tf.shape(logits)[1]

        # --- Emit score ---
        batch_indices = tf.tile(tf.expand_dims(tf.range(batch_size), 1),
[1, seq_len])
        time_indices = tf.tile(tf.expand_dims(tf.range(seq_len), 0),
[batch_size, 1])
        gather_indices = tf.stack([batch_indices, time_indices, tags],
axis=-1)
        emit_score = tf.gather_nd(logits, gather_indices)
        emit_score = tf.reduce_sum(emit_score * mask, axis=1)

        # --- Transition score ---
        trans_score = 0.0
        for t in range(seq_len - 1):
            cur_tag = tags[:, t]
            next_tag = tags[:, t + 1]
            pair_score = tf.gather_nd(self.transitions,
tf.stack([cur_tag, next_tag], axis=1))
```

```
            trans_score += pair_score * mask[:, t + 1]

        log_likelihood = emit_score + trans_score - log_Z
        return -tf.reduce_mean(log_likelihood)

    def viterbi_decode(self, logits):
        seq_len = tf.shape(logits)[1]
        dp = logits[:, 0]
        paths = tf.cast(tf.argmax(dp, axis=1), tf.int32)[:, None]
        for t in range(1, seq_len):
            emit = logits[:, t]
            scores = tf.expand_dims(dp, 2) + self.transitions
            dp = tf.reduce_max(scores, axis=1) + emit
            step_best = tf.cast(tf.argmax(dp, axis=1), tf.int32)[:,
None]
            paths = tf.concat([paths, step_best], axis=1)
        return paths


# ========== 4. Model ==========
class BiLSTM_CRF(Model):
    def __init__(self, vocab_size, num_tags):
        super().__init__()
        self.emb = layers.Embedding(vocab_size, 64, mask_zero=True)
        self.bilstm = layers.Bidirectional(layers.LSTM(64,
return_sequences=True))
        self.dense = layers.Dense(num_tags)
        self.crf = CRFLayer(num_tags)

    def call(self, inputs, training=False):
        x = self.emb(inputs)
        x = self.bilstm(x)
        logits = self.dense(x)
        return logits


# ========== 5. Train ==========
model = BiLSTM_CRF(vocab_size, num_tags)
_ = model(tf.constant(X))
model.crf.build((None, max_len, num_tags))
optimizer = tf.keras.optimizers.Adam(0.01)

for epoch in range(100):
    with tf.GradientTape() as tape:
        logits = model(X, training=True)
        loss = model.crf.loss(logits, y, mask)
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
    if epoch % 10 == 0:
        print(f"Epoch {epoch}, Loss: {loss.numpy():.4f}")

# ========== 6. Predict ==========
logits = model(X)
pred_tags = model.crf.viterbi_decode(logits).numpy()

for i, sent in enumerate(sentences):
    print(f"\nSentence: {' '.join(sent)}")
    print("True Tags:      ", [idx2tag[t] for t in y[i][:len(sent)]])
    print("Predicted Tags:", [idx2tag[t] for t in pred_tags[i]
[:len(sent)]])
```

OUTPUT:



```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/la
  warnings.warn(
Epoch 0, Loss: 6.7208
Epoch 10, Loss: 0.9994
Epoch 20, Loss: 0.0127
Epoch 30, Loss: 0.0014
Epoch 40, Loss: 0.0005
Epoch 50, Loss: 0.0003
Epoch 60, Loss: 0.0003
Epoch 70, Loss: 0.0002
Epoch 80, Loss: 0.0002
Epoch 90, Loss: 0.0002

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

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

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

LEARNING OUTCOME: