EXPERIMENT – 01

Aim: Setting up the Spyder IDE Environment and Executing a Python Program

Theory:

Spyder IDE Overview:

- **Spyder (Scientific Python Development Environment)** is an open-source IDE specifically designed for Python. It provides a user-friendly interface that includes an editor, console, variable explorer, and other features that facilitate data analysis and scientific programming.

Key Features:

- **Editor**: Write and edit Python scripts with syntax highlighting.
- **Console**: Execute code interactively and view output.
- **Variable Explorer**: Inspect and modify variables in the workspace.
- **Integrated IPython**: Enhanced interactive Python shell with support for inline plotting.

Setting Up Spyder IDE

1. **Download and Install Anaconda**:
    - Visit the Anaconda website and download the Anaconda distribution suitable for your operating system.
    - Follow the installation instructions provided on the site.
2. **Open Anaconda Navigator**:
    - Once Anaconda is installed, open Anaconda Navigator from your applications.
3. **Launch Spyder**:
    - In Anaconda Navigator, find Spyder in the list of available applications and click "Launch".
4. **Configure Spyder** (Optional):
    - You can customize Spyder's interface through the "Preferences" menu. This includes changing themes, configuring keyboard shortcuts, and adjusting console settings.

Writing and Executing a Python Program

Example Program: Hello, World!

1. **Create a New Python File**:
    - In Spyder, click on **File**>**New File** to open a new editor tab.

2. **Write the Code**:
    - Enter the following Python code in the editor:

.

Code:

num1 = 10

num2 = 5

sum_result = num1 + num2

print("The sum of", num1, "and", num2, "is:", sum_result)

Output:

```
In [68]: runfile('C:/College/RLDL Lab/untitled0.py',
wdir='C:/College/RLDL Lab')
The sum of 10 and 5 is: 15
```

EXPERIMENT – 02

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

Theory:

Keras

- **Keras** is a high-level neural networks API written in Python. It is designed to enable fast experimentation and is user-friendly, modular, and extensible. Keras can run on top of various deep learning frameworks, but it's most commonly used with TensorFlow.
- **Key Features**:
    - Simplifies building and training neural networks.
    - Supports convolutional and recurrent networks as well as combinations of both.
    - Provides tools for data preprocessing and augmentation.

TensorFlow

- **TensorFlow** is an open-source deep learning framework developed by Google. It is used for numerical computation and machine learning, and it allows developers to create complex deep learning models.
- **Key Features**:
    - Offers a flexible architecture that can run on various platforms (CPUs, GPUs, TPUs).
    - Supports large-scale machine learning and is equipped with features for distributed training.
    - Provides a comprehensive ecosystem, including TensorBoard for visualization and TensorFlow Lite for mobile and embedded devices.

PyTorch

- **PyTorch** is an open-source deep learning framework developed by Facebook's AI Research lab. It is known for its dynamic computation graph, which makes it easier to build and modify neural networks on the fly.
- **Key Features**:
    - Provides a more Pythonic and intuitive interface, which is popular among researchers.
    - Supports GPU acceleration and has strong integration with NumPy.
    - Includes a rich set of libraries and tools for computer vision (torchvision), natural language processing (torchtext), and more.

Code 1:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import mnist

# Load and prepare the data
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

# Build the model
model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])

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

# Train the model
model.fit(x_train, y_train, epochs=5)

# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f'Test accuracy: {test_acc}')
```

Output 1:

```
In [69]: runfile('C:/College/RLDL Lab/Exp2.py', wdir='C:/College/RLDL Lab')
Epoch 1/5
1875/1875 [==============================] - 3s 1ms/step - loss: 0.2575 - accuracy: 0.9263
Epoch 2/5
1875/1875 [==============================] - 3s 1ms/step - loss: 0.1142 - accuracy: 0.9668
Epoch 3/5
1875/1875 [==============================] - 3s 1ms/step - loss: 0.0784 - accuracy: 0.9760
Epoch 4/5
1875/1875 [==============================] - 3s 1ms/step - loss: 0.0582 - accuracy: 0.9825
Epoch 5/5
1875/1875 [==============================] - 3s 1ms/step - loss: 0.0449 - accuracy: 0.9866
313/313 - 0s - loss: 0.0799 - accuracy: 0.9763 - 371ms/epoch - 1ms/step
Test accuracy: 0.9763000011444092
```

Code 2:

```
import torch

import torch.nn as nn

import torch.optim as optim

import torchvision

import torchvision.transforms as transforms

from torch.utils.data import DataLoader


# Data preparation

transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])

train_set = torchvision.datasets.MNIST(root='./data', train=True, download=True,
transform=transform)

test_set = torchvision.datasets.MNIST(root='./data', train=False, download=True,
transform=transform)

train_loader = DataLoader(train_set, batch_size=64, shuffle=True)

test_loader = DataLoader(test_set, batch_size=64, shuffle=False)


# Model definition

class SimpleNN(nn.Module):

    def __init__(self):

        super(SimpleNN, self).__init__()
```

```
self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(28*28, 128)
        self.fc2 = nn.Linear(128, 10)

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

# Instantiate model, define loss and optimizer
model = SimpleNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

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

# Evaluation
correct = 0
total = 0
```

```
with torch.no_grad():
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy: {100 * correct / total}%')
```

Output 2:

```
In [70]: runfile('C:/College/RLDL Lab/Exp2b.py', wdir='C:/College/RLDL Lab')
Epoch 1, Loss: 0.16297198832035065
Epoch 2, Loss: 0.22283615171909332
Epoch 3, Loss: 0.05125654488801956
Epoch 4, Loss: 0.26888200640678406
Epoch 5, Loss: 0.28956353664398193
Accuracy: 96.94%
```

EXPERIMENT – 03

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:

Introduction to Reinforcement Learning (RL)

- **Reinforcement Learning (RL)** is a subfield of machine learning where an agent learns to make decisions by interacting with an environment. The goal of the agent is to maximize cumulative rewards by learning the best actions to take in various states of the environment.
- **Key Concepts**:
    - **Agent**: The learner or decision-maker.
    - **Environment**: The world the agent interacts with.
    - **State (s)**: A representation of the environment at a specific time.
    - **Action (a)**: A choice made by the agent that affects the state.
    - **Reward (r)**: Feedback from the environment based on the action taken.
    - **Policy (π)**: A strategy that defines the agent's way of behaving at a given time.
    - **Q-value (Q)**: Represents the expected utility of taking a given action in a given state.

Q-Learning

- **Q-learning** is a value-based off-policy reinforcement learning algorithm that aims to learn the value of an action in a particular state. The "Q" in Q-learning stands for "quality."
- **Objective**: The objective of Q-learning is to learn a policy that maximizes the expected cumulative reward by estimating the Q-values for state-action pairs.
- **Q-Learning Algorithm**:
    - Initialize the Q-table with zeros for all state-action pairs.
    - For each episode:
        - Initialize the state.
        - For each step in the episode:

- Choose an action based on the current state using an exploration strategy (like ε-greedy).
- Take the action and observe the reward and the new state.
- Update the Q-value using the Q-learning update rule: Q(s,a)←Q(s,a)+α[r+γmax a′Q(s′,a′)−Q(s,a)]Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]Q(s,a)←Q(s,a)+α[r+γa′maxQ(s′,a′)−Q(s,a)]
- Set the new state as the current state.

  - The parameters involved are:
    - α\alphaα (learning rate): Determines how much of the new Q-value to incorporate into the existing Q-value.
    - γ\gammaγ (discount factor): Determines the importance of future rewards.

Source Code:

```
import numpy as np

import gym

from gym import spaces


# Custom Grid World Environment

class GridWorld(gym.Env):

   def __init__(self):

      super(GridWorld, self).__init__()

self.grid_size = 5  # 5x5 grid

self.start_pos = (0, 0)

self.goal_pos = (4, 4)

self.state = self.start_pos


      # Define action and observation space

self.action_space = spaces.Discrete(4)  # 0: up, 1: down, 2: left, 3: right

self.observation_space = spaces.Discrete(self.grid_size * self.grid_size)
```

```
    def reset(self):
self.state = self.start_pos
        return self._get_state_index(self.state)

    def step(self, action):
        if action == 0 and self.state[0] > 0:  # Move up
self.state = (self.state[0] - 1, self.state[1])
elif action == 1 and self.state[0] <self.grid_size - 1:  # Move down
self.state = (self.state[0] + 1, self.state[1])
elif action == 2 and self.state[1] > 0:  # Move left
self.state = (self.state[0], self.state[1] - 1)
elif action == 3 and self.state[1] <self.grid_size - 1:  # Move right
self.state = (self.state[0], self.state[1] + 1)

        # Check if the agent reached the goal
        done = self.state == self.goal_pos
        reward = 1 if done else -0.01  # Reward for reaching the goal, small penalty otherwise
        return self._get_state_index(self.state), reward, done, {}

    def _get_state_index(self, state):
        return state[0] * self.grid_size + state[1]  # Convert 2D state to 1D index

    def render(self):
        grid = np.zeros((self.grid_size, self.grid_size), dtype=str)
        grid[:] = '.'
        grid[self.goal_pos] = 'G'  # Goal
        grid[self.state] = 'A'  # Agent
        print("\n".join(" ".join(row) for row in grid))
```

```
# Q-learning algorithm
def q_learning(env, episodes=1000, learning_rate=0.1, discount_factor=0.9,
exploration_rate=1.0, exploration_decay=0.995, exploration_min=0.01):
Q_table = np.zeros((env.observation_space.n, env.action_space.n))

    for episode in range(episodes):
        state = env.reset()
        done = False

        while not done:
            # Choose action: Explore or Exploit
            if np.random.rand() <exploration_rate:
                action = env.action_space.sample()  # Random action (explore)
            else:
                action = np.argmax(Q_table[state])  # Best action (exploit)

            # Take action and observe the reward and new state
new_state, reward, done, _ = env.step(action)

            # Update Q-value
Q_table[state, action] = (1 - learning_rate) * Q_table[state, action] + \
learning_rate * (reward + discount_factor * np.max(Q_table[new_state]))

            # Update state
            state = new_state

        # Decay the exploration rate
exploration_rate = max(exploration_min, exploration_rate * exploration_decay)
```

```
        # Optional: Print progress every 100 episodes
        if (episode + 1) % 100 == 0:
            print(f"Episode: {episode + 1}")

    print("Training finished.\n")
    return Q_table

# Watch the trained agent play the game
def watch_agent(env, Q_table):
    state = env.reset()
    done = False
env.render()

    while not done:
        action = np.argmax(Q_table[state])  # Choose best action based on Q-table
        state, reward, done, _ = env.step(action)  # Take action
env.render()

# Main execution
if __name__ == "__main__":
    # Create the environment
    env = GridWorld()

    # Train the agent using Q-learning
Q_table = q_learning(env, episodes=1000)

    # Watch the trained agent
```

watch_agent(env, Q_table)

Output:

```
In [71]: runfile('C:/College/RLDL Lab/Exp3.py', wdir='C:/College/RLDL Lab')
Episode: 100
Episode: 200
Episode: 300
Episode: 400
Episode: 500
Episode: 600
Episode: 700
Episode: 800
Episode: 900
Episode: 1000
Training finished.
```

```
A . . . .
. . . . .
. . . . .
. . . . .
. . . . G
. . . . .
A . . . .
. . . . .
. . . . .
. . . . G
. . . . .
. . . . .
A . . . .
. . . . .
. . . . G
```

```
. . . . .
. . . . .
. . . . A
. . . . .
. . . . G
. . . . .
. . . . .
. . . . .
. . . . A
. . . . G
. . . . .
. . . . .
. . . . .
. . . . .
. . . . A
```

EXPERIMENT – 04

Aim: Implement deep Q-network with PyTorch

Theory:

Overview of Deep Q-Networks (DQN)

- **Deep Q-Network (DQN)** is an extension of Q-learning that uses a neural network to approximate the Q-value function. This approach was developed by DeepMind and successfully applied to play Atari games.
- **Purpose**: In traditional Q-learning, we maintain a Q-table to store Q-values for each state-action pair. However, when the state space is large (e.g., images in Atari games), storing and updating a Q-table becomes impractical. DQNs solve this by using a deep neural network to approximate Q-values.

Key Concepts in DQN

1. **Q-Learning Recap**:
    - Q-learning seeks to learn an optimal policy by updating the Q-values according to: Q(s,a)←Q(s,a)+α[r+γmax a′Q(s′,a′)−Q(s,a)]Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]Q(s,a)←Q(s,a)+α[r+γa′maxQ(s′,a′)−Q(s,a)]
    - The DQN replaces the Q-table with a neural network that estimates Q(s,a)Q(s, a)Q(s,a).
2. **Neural Network as Q-Function**:
    - The DQN algorithm leverages a neural network that takes the current state as input and outputs Q-values for all possible actions in that state. This is particularly helpful when the state space is continuous or too large for a table-based approach.
3. **Experience Replay**:
    - **Experience Replay** is a technique where we store the agent's experiences (state, action, reward, next state, done) in a buffer (replay memory).
    - During training, we randomly sample mini-batches of experiences from this buffer to update the neural network. This approach reduces the correlation between consecutive experiences and increases data efficiency.
4. **Target Network**:
    - DQN uses a separate **target network**, which is a copy of the Q-network. The target network is used to calculate the target Q-value during training, while the Q-network is updated at each step.

- The target network is updated with the Q-network's weights periodically, which helps to stabilize training by preventing rapid changes in the target Q-values.

5. **Exploration vs. Exploitation**:
    - To balance exploration (trying new actions) and exploitation (choosing the best-known action), DQN uses an **ε-greedy policy**. The agent randomly chooses an action with probability ε and selects the action with the highest Q-value otherwise. The value of ε is usually decreased over time to shift from exploration to exploitation.

DQN Algorithm Summary

1. Initialize the Q-network and target network with random weights.
2. Initialize replay memory to store experiences.
3. For each episode:
    - Start with the initial state.
    - For each step in the episode:
        - Choose an action using the ε-greedy policy.
        - Take the action, observe reward, and transition to the next state.
        - Store the experience in the replay memory.
        - Sample a random batch of experiences from the replay memory.
        - For each experience, compute the target Q-value.
        - Update the Q-network by minimizing the mean squared error between the predicted and target Q-values.
        - Periodically update the target network.

Implementing DQN in PyTorch

Here's a basic outline of how to implement DQN using PyTorch.

1. **Set Up the Environment**: Use OpenAI Gym to provide an environment for the agent to interact with.
2. **Define the Q-Network**: Create a neural network that takes the state as input and outputs Q-values for each action.
3. **Implement Experience Replay**: Set up a replay memory to store and sample experiences.
4. **Train the Agent**: Implement the DQN algorithm to train the agent, including updating the target network periodically.

Code:

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from collections import deque
import random
import gym
import matplotlib.pyplot as plt

# [Previous DQN, ReplayBuffer, and DQNAgent classes remain the same]
# ... [Keep all the class implementations exactly as they were before]
class DQN(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_dim=128):
        """
        Initialize Deep Q-Network
input_dim: number of input features
output_dim: number of possible actions
hidden_dim: size of hidden layers
        """
        super(DQN, self).__init__()

self.network = nn.Sequential(
nn.Linear(input_dim, hidden_dim),
nn.ReLU(),
nn.Linear(hidden_dim, hidden_dim),
nn.ReLU(),
nn.Linear(hidden_dim, output_dim)
```

```
        )

    def forward(self, x):
        return self.network(x)

class ReplayBuffer:
    def __init__(self, capacity):
        """
        Initialize Replay Buffer
        capacity: maximum size of buffer
        """
self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        """Add experience to buffer"""
self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        """Sample random batch of experiences"""
        state, action, reward, next_state, done = zip(*random.sample(self.buffer, batch_size))
        return (torch.FloatTensor(state),
torch.LongTensor(action),
torch.FloatTensor(reward),
torch.FloatTensor(next_state),
torch.FloatTensor(done))

    def __len__(self):
        return len(self.buffer)
```

```
class DQNAgent:
    def __init__(self, state_dim, action_dim, hidden_dim=128, lr=1e-3, gamma=0.99,
epsilon_start=1.0, epsilon_end=0.01, epsilon_decay=0.995,
buffer_size=10000, batch_size=64, target_update=10):
        """
        Initialize DQN Agent
state_dim: dimension of state space
action_dim: dimension of action space
hidden_dim: size of hidden layers
lr: learning rate
        gamma: discount factor
        epsilon_*: exploration parameters
buffer_size: size of replay buffer
batch_size: size of training batch
target_update: frequency of target network update
        """
self.action_dim = action_dim
self.gamma = gamma
self.epsilon = epsilon_start
self.epsilon_end = epsilon_end
self.epsilon_decay = epsilon_decay
self.batch_size = batch_size
self.target_update = target_update

        # Networks
self.policy_net = DQN(state_dim, action_dim, hidden_dim)
self.target_net = DQN(state_dim, action_dim, hidden_dim)
```

```
self.target_net.load_state_dict(self.policy_net.state_dict())

self.optimizer = optim.Adam(self.policy_net.parameters(), lr=lr)
self.memory = ReplayBuffer(buffer_size)

self.steps = 0

    def select_action(self, state):
        """Epsilon-greedy action selection"""
        if random.random() >self.epsilon:
            with torch.no_grad():
                state = torch.FloatTensor(state).unsqueeze(0)
q_values = self.policy_net(state)
                return q_values.max(1)[1].item()
        else:
            return random.randrange(self.action_dim)

    def update(self):
        """Update network weights"""
        if len(self.memory) <self.batch_size:
            return

        # Sample batch and compute Q values
        state, action, reward, next_state, done = self.memory.sample(self.batch_size)

current_q = self.policy_net(state).gather(1, action.unsqueeze(1))
next_q = self.target_net(next_state).max(1)[0].detach()
target_q = reward + (1 - done) * self.gamma * next_q
```

```
        # Compute loss and update weights
        loss = nn.MSELoss()(current_q.squeeze(), target_q)

self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()

        # Update target network
        if self.steps % self.target_update == 0:
self.target_net.load_state_dict(self.policy_net.state_dict())

        # Update epsilon
self.epsilon = max(self.epsilon_end, self.epsilon * self.epsilon_decay)
self.steps += 1

        return loss.item()

def train_dqn(env, agent, episodes, max_steps=1000):
    """
    Train DQN agent
    env: gym environment
    agent: DQNAgent instance
    episodes: number of training episodes
max_steps: maximum steps per episode
    """
    rewards = []
```

```
    for episode in range(episodes):
        state = env.reset()
episode_reward = 0

        for step in range(max_steps):
            # Select and perform action
            action = agent.select_action(state)
next_state, reward, done, _ = env.step(action)

            # Store transition
agent.memory.push(state, action, reward, next_state, done)

            # Update network
            loss = agent.update()

episode_reward += reward
            state = next_state

            if done:
                break

rewards.append(episode_reward)

        # Print progress
        if (episode + 1) % 10 == 0:
avg_reward = np.mean(rewards[-10:])
            print(f"Episode {episode + 1}, Average Reward: {avg_reward:.2f}, Epsilon:
{agent.epsilon:.2f}")
```

```
    return rewards

def train_dqn(env, agent, episodes, max_steps=1000):
    """
    Train DQN agent
    env: gym environment
    agent: DQNAgent instance
    episodes: number of training episodes
max_steps: maximum steps per episode
    """
    rewards = []

    for episode in range(episodes):
        state, _ = env.reset()  # Modified to handle new gym API
episode_reward = 0

        for step in range(max_steps):
            # Select and perform action
            action = agent.select_action(state)
next_state, reward, done, _, _ = env.step(action)  # Modified to handle new gym API

            # Store transition
agent.memory.push(state, action, reward, next_state, done)

            # Update network
            loss = agent.update()

episode_reward += reward
```

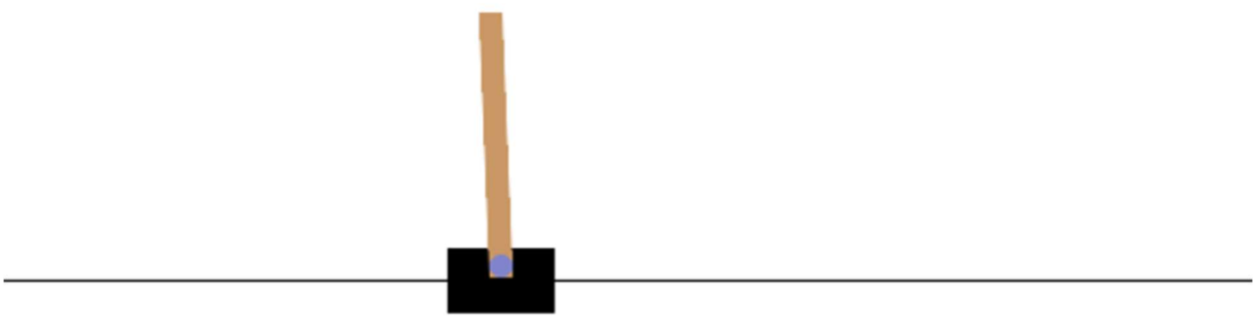
```
            state = next_state

            if done:
                break

rewards.append(episode_reward)

        # Print progress
        if (episode + 1) % 10 == 0:
avg_reward = np.mean(rewards[-10:])
            print(f"Episode {episode + 1}, Average Reward: {avg_reward:.2f}, Epsilon:
{agent.epsilon:.2f}")

    return rewards

def plot_rewards(rewards):
    """Plot the training rewards"""
plt.figure(figsize=(10, 5))
plt.plot(rewards)
plt.title('Training Rewards')
plt.xlabel('Episode')
plt.ylabel('Reward')
plt.show()

def test_agent(env, agent, episodes=10, render=True):
    """Test the trained agent"""
    for episode in range(episodes):
        state, _ = env.reset()  # Modified to handle new gym API
total_reward = 0
```

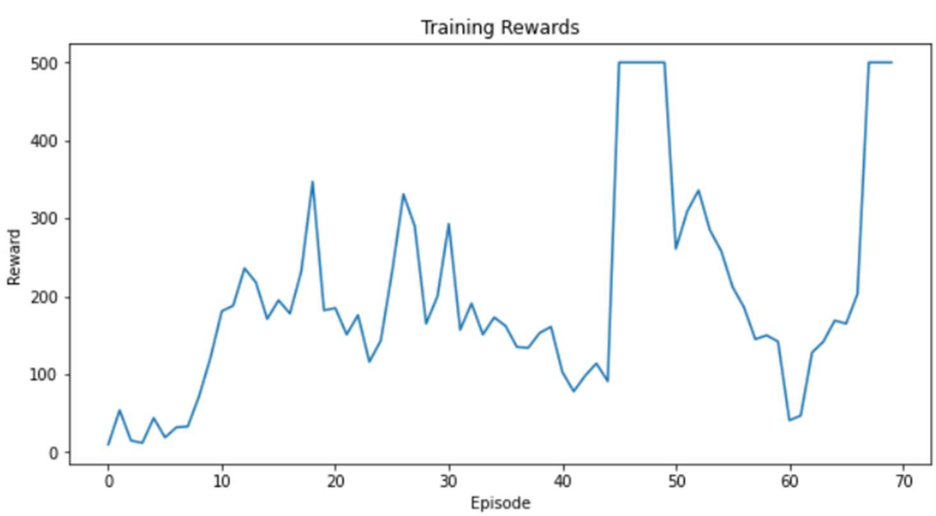
```
        done = False

        while not done:
            if render:
env.render()

            # Select action without exploration
            with torch.no_grad():
state_tensor = torch.FloatTensor(state).unsqueeze(0)
                action = agent.policy_net(state_tensor).max(1)[1].item()

            state, reward, done, _, _ = env.step(action)  # Modified to handle new gym API
total_reward += reward

        print(f"Test Episode {episode + 1}: Total Reward: {total_reward}")

env.close()

if __name__ == "__main__":
    # Set random seeds for reproducibility
random.seed(42)
np.random.seed(42)
torch.manual_seed(42)

    # Create environment
    env = gym.make('CartPole-v1', render_mode="human")  # Modified to specify render mode

    # Get environment dimensions
```

```
state_dim = env.observation_space.shape[0]  # 4 for CartPole
action_dim = env.action_space.n  # 2 for CartPole

    # Initialize agent
    agent = DQNAgent(
state_dim=state_dim,
action_dim=action_dim,
hidden_dim=128,
lr=1e-3,
        gamma=0.99,
epsilon_start=1.0,
epsilon_end=0.01,
epsilon_decay=0.995,
buffer_size=10000,
batch_size=64,
target_update=10
    )

    # Train the agent
    print("Starting training...")
    rewards = train_dqn(env, agent, episodes=70, max_steps=500)

    # Plot training rewards
plot_rewards(rewards)

    # Test the trained agent
    print("\nTesting the trained agent...")
test_agent(env, agent, episodes=5, render=True)
```

```
    # Save the trained model
torch.save(agent.policy_net.state_dict(), 'dqn_cartpole.pth')
    print("\nModel saved to 'dqn_cartpole.pth'")
```

Output:





```
Starting training...
Episode 10, Average Reward: 17.00, Epsilon: 0.58
Episode 20, Average Reward: 41.50, Epsilon: 0.07
Episode 30, Average Reward: 195.70, Epsilon: 0.01
Episode 40, Average Reward: 203.20, Epsilon: 0.01
Episode 50, Average Reward: 202.20, Epsilon: 0.01
Episode 60, Average Reward: 191.50, Epsilon: 0.01
Episode 70, Average Reward: 187.30, Epsilon: 0.01

Testing the trained agent...
Test Episode 1: Total Reward: 229.0
Test Episode 2: Total Reward: 211.0
Test Episode 3: Total Reward: 253.0
Test Episode 4: Total Reward: 293.0
Test Episode 5: Total Reward: 228.0
```

EXPERIMENT – 05

Aim: Python implementation of the iterative policy evaluation and update.

Theory:

Iterative Policy Evaluation and Update is part of the **Policy Iteration** algorithm in **Reinforcement Learning (RL)**, which is used to find an optimal policy for a Markov Decision Process (MDP). Policy Iteration includes two main steps:

1. **Policy Evaluation**: Calculate the value function for a given policy.
2. **Policy Improvement**: Update the policy by making it greedy with respect to the current value function.

Markov Decision Process (MDP) Recap

- An MDP is defined by the tuple (S,A,P,R,γ)(S, A, P, R, \gamma)(S,A,P,R,γ):
    - **S**: Set of possible states.
    - **A**: Set of possible actions.
    - **P**: Transition probability, where P(s′|s,a)P(s'|s, a)P(s′|s,a) is the probability of transitioning to state s′s's′ from state sss after taking action aaa.
    - **R**: Reward function, where R(s,a,s′)R(s, a, s')R(s,a,s′) is the reward received after moving from sss to s′s's′ via aaa.
    - **γ**: Discount factor, representing the weight given to future rewards (ranges from 0 to 1).

Policy Evaluation

- In Policy Evaluation, we estimate the value function V(s)V(s)V(s) for a given policy π\piπ.
- The **value function** Vπ(s)V^\pi(s)Vπ(s) represents the expected cumulative reward the agent will receive from state sss following policy π\piπ.
- **Bellman Expectation Equation** for Policy Evaluation:
  V(s)=∑aπ(a|s)∑s′P(s′|s,a)[R(s,a,s′)+γV(s′)]V(s) = \sum_{a} \pi(a|s) \sum_{s'} P(s'|s, a) \left[R(s, a, s') + \gamma V(s') \right]V(s)=a∑π(a|s)s′∑P(s′|s,a)[R(s,a,s′)+γV(s′)]
- This is an iterative process where we update V(s)V(s)V(s) for each state sss until V(s)V(s)V(s) converges to the true value for the given policy π\piπ.

Policy Improvement

- **Policy Improvement** is where we update the policy based on the current value function V(s)V(s)V(s).

- For each state sss, we make the policy greedy by choosing actions that maximize the expected return.
- The new policy π′\pi'π′ is: π′(s)=argmaxa∑s′P(s′|s,a)[R(s,a,s′)+γV(s′)]\pi'(s) = \text{argmax}_a \sum_{s'} P(s'|s, a) \left[R(s, a, s') + \gamma V(s') \right]π′(s)=argmaxa s′∑P(s′|s,a)[R(s,a,s′)+γV(s′)]
- This greedy approach with respect to V(s)V(s)V(s) ensures that the policy becomes increasingly better.

Policy Iteration Algorithm

- The complete **Policy Iteration** algorithm alternates between Policy Evaluation and Policy Improvement until the policy is stable (i.e., no longer changes).

1. **Initialize** the policy π\piπ and value function V(s)V(s)V(s).
2. **Policy Evaluation**: Update V(s)V(s)V(s) until it converges.
3. **Policy Improvement**: Update the policy by making it greedy with respect to V(s)V(s)V(s).
4. Repeat steps 2 and 3 until the policy stops changing.

Code:

```
import numpy as np


class PolicyIteration:
    def __init__(self, states, actions, transitions, rewards, gamma=0.9, theta=1e-6):
        """
        Initialize the Policy Iteration algorithm.


Args:
            states: Number of states in the MDP
            actions: Number of actions in the MDP
            transitions: 3D array [s, a, s'] containing transition probabilities
            rewards: 2D array [s, a] containing immediate rewards
            gamma: Discount factor
```

```
            theta: Convergence threshold for policy evaluation
        """
self.states = states
self.actions = actions
self.transitions = transitions
self.rewards = rewards
self.gamma = gamma
self.theta = theta

        # Initialize random policy
self.policy = np.random.randint(0, actions, size=states)
self.value_function = np.zeros(states)

    def policy_evaluation(self):
        """
        Evaluate the current policy using iterative policy evaluation.
        """
        while True:
            delta = 0
            for s in range(self.states):
                v = self.value_function[s]

                # Calculate new state value
                a = self.policy[s]
new_v = 0

                for next_s in range(self.states):
new_v += self.transitions[s, a, next_s] * (
```

```
self.rewards[s, a] +
self.gamma * self.value_function[next_s]
                    )

self.value_function[s] = new_v
                delta = max(delta, abs(v - new_v))

            if delta <self.theta:
                break

    def policy_improvement(self):
        """
        Improve the current policy based on the value function.

        Returns:
            bool: True if policy changed, False otherwise
        """
policy_stable = True

        for s in range(self.states):
old_action = self.policy[s]
action_values = np.zeros(self.actions)

            # Calculate value for each action
            for a in range(self.actions):
                for next_s in range(self.states):
action_values[a] += self.transitions[s, a, next_s] * (
self.rewards[s, a] +
```

```
self.gamma * self.value_function[next_s]
                    )

            # Choose best action
self.policy[s] = np.argmax(action_values)

            if old_action != self.policy[s]:
policy_stable = False

        return policy_stable

    def run(self, max_iterations=1000):
        """
        Run the complete policy iteration algorithm.

Args:
max_iterations: Maximum number of iterations to run

        Returns:
            tuple: (optimal policy, optimal value function)
        """
        for i in range(max_iterations):
            # 1. Policy Evaluation
self.policy_evaluation()

            # 2. Policy Improvement
policy_stable = self.policy_improvement()
```

```
            if policy_stable:
                print(f"Policy converged after {i+1} iterations")
                break

        return self.policy, self.value_function

# Driver code with a simple grid world example
def create_grid_world(size=4):
    """
    Create a simple grid world MDP.
    States are numbered from 0 to size^2-1.
    Actions are: 0=up, 1=right, 2=down, 3=left
    """
n_states = size * size
n_actions = 4

    # Initialize transitions and rewards
    transitions = np.zeros((n_states, n_actions, n_states))
    rewards = np.zeros((n_states, n_actions))

    # Set goal state (top-right corner) and penalty state (bottom-right corner)
goal_state = size - 1
penalty_state = size * size - 1

    # Fill transition probabilities and rewards
    for s in range(n_states):
        if s == goal_state or s == penalty_state:
            continue
```

```
        row, col = s // size, s % size

        for a in range(n_actions):
            # Calculate next state based on action
next_row, next_col = row, col

            if a == 0:  # up
next_row = max(0, row - 1)
elif a == 1:  # right
next_col = min(size - 1, col + 1)
elif a == 2:  # down
next_row = min(size - 1, row + 1)
elif a == 3:  # left
next_col = max(0, col - 1)

next_s = next_row * size + next_col

            # Set transition probability (0.8 for intended direction, 0.1 for adjacent directions)
            transitions[s, a, next_s] = 0.8

            # Add small probability of moving sideways
            for noise_a in [(a-1)%4, (a+1)%4]:
noise_row, noise_col = row, col

                if noise_a == 0:
noise_row = max(0, row - 1)
elifnoise_a == 1:
```

```
noise_col = min(size - 1, col + 1)
elifnoise_a == 2:
noise_row = min(size - 1, row + 1)
elifnoise_a == 3:
noise_col = max(0, col - 1)

noise_s = noise_row * size + noise_col
        transitions[s, a, noise_s] = 0.1

      # Set rewards
      if next_s == goal_state:
        rewards[s, a] = 1.0
elifnext_s == penalty_state:
        rewards[s, a] = -1.0
      else:
        rewards[s, a] = -0.04  # Small negative reward for each step

  return transitions, rewards

def main():
  # Create a 4x4 grid world
  size = 4
  transitions, rewards = create_grid_world(size)

  # Initialize and run policy iteration
  pi = PolicyIteration(
    states=size*size,
    actions=4,
```

```
        transitions=transitions,
        rewards=rewards,
        gamma=0.9,
        theta=1e-6
    )

optimal_policy, optimal_values = pi.run()

    # Print results
    print("\nOptimal Policy (0=up, 1=right, 2=down, 3=left):")
    print(optimal_policy.reshape(size, size))

    print("\nOptimal Value Function:")
    print(np.round(optimal_values.reshape(size, size), 3))

if __name__ == "__main__":
    main()
```

Output:

```
Policy converged after 5 iterations

Optimal Policy (0=up, 1=right, 2=down, 3=left):
[[1 1 1 0]
 [1 1 1 0]
 [0 0 0 0]
 [0 0 0 0]]

Optimal Value Function:
[[0.804 0.987 1.198 0.   ]
 [0.684 0.835 1.006 1.198]
 [0.565 0.687 0.835 0.987]
 [0.458 0.551 0.611 0.   ]]
```

EXPERIMENT – 06

Aim: Chatbot using bi-directional LSTMs

Theory:

Chatbot Fundamentals

A chatbot is an application designed to simulate human conversation by processing user inputs and generating appropriate responses. Chatbots typically rely on:

1. **Natural Language Understanding (NLU)**: Interpreting user inputs.
2. **Natural Language Generation (NLG)**: Formulating responses.

Two common chatbot approaches include:

- **Rule-Based Chatbots**: Rely on pre-defined responses for specific keywords or phrases.
- **Machine Learning-Based Chatbots**: Use deep learning or NLP models to learn from large conversational datasets and generate dynamic responses.

For complex, flexible conversation, machine learning-based chatbots are more effective. Bi-LSTM models are well-suited for understanding context, which is vital in these more advanced chatbots.

Recurrent Neural Networks (RNN) and LSTMs

- **Recurrent Neural Networks (RNNs)**: Designed to process sequences of data by passing the output from one time step as input to the next. However, RNNs suffer from the **vanishing gradient problem**, making it hard to learn long-term dependencies.
- **Long Short-Term Memory Networks (LSTMs)**: An improvement over RNNs, LSTMs are designed to capture long-term dependencies by using a memory cell and gating mechanisms (input, output, and forget gates). This makes LSTMs effective for handling sequential data like sentences.

Bi-Directional LSTMs (Bi-LSTMs)

- **Bi-Directional LSTMs** are a variant of LSTMs that process input sequences in both forward and backward directions. Each time step has two hidden states: one moving forward (left to right) and the other moving backward (right to left).
- **Benefit**: This structure allows Bi-LSTMs to capture contextual information from both the past (previous words) and the future (upcoming words) within a sentence, giving them a richer understanding of the sentence context.

For instance, in understanding the phrase “I live in New York,” a Bi-LSTM considers both the words before and after each term, making it better equipped to understand that “New York” is a location, given context.

Architecture for a Bi-LSTM-Based Chatbot

To create a chatbot using Bi-LSTM, we typically use a sequence-to-sequence (Seq2Seq) model, often paired with an attention mechanism for better context handling. Here's how the components work together:

1. **Encoder**:
   - The encoder processes the input sentence (e.g., user query) using a Bi-LSTM. Each word in the sentence is passed through the Bi-LSTM, which generates a hidden state for each time step.
   - By using both forward and backward LSTM layers, the encoder creates a comprehensive context vector, which summarizes the input sentence.
2. **Attention Mechanism (Optional but Beneficial)**:
   - An attention mechanism allows the model to focus on relevant parts of the input sequence when generating each word in the output. This is especially useful in longer sentences where all parts of the sentence may not be equally important for generating a response.
3. **Decoder**:
   - The decoder is a unidirectional LSTM or Bi-LSTM that generates the response, one word at a time, using the context vector from the encoder.
   - It can use the attention mechanism to dynamically weigh which parts of the input sequence to focus on at each step of the output generation.
4. **Output Layer**:
   - The decoder's hidden states are passed through a softmax layer to produce probabilities over the vocabulary for each output word, allowing the model to generate a response word-by-word.

Training Process

- **Data Preparation**: Prepare a conversational dataset (e.g., questions and answers or prompts and responses) and convert words into numerical representations, such as word embeddings (e.g., GloVe, Word2Vec, or BERT embeddings).
- **Training Objective**: Minimize the difference between predicted and actual responses. This is usually done by minimizing cross-entropy loss, which measures the accuracy of the predicted word distributions.
- **Training with Teacher Forcing**: During training, the actual next word is often fed as input to the decoder rather than its own previous prediction, a process called "teacher

forcing." This helps the model learn more accurately by providing a reliable input sequence during training.

Bi-LSTM's Role in Chatbots

- **Contextual Understanding**: Because Bi-LSTMs consider both past and future words, they are better at capturing the entire context of a sentence, improving response accuracy.
- **Handling Complex Queries**: Bi-LSTMs are effective in understanding complex sentences where the meaning of each word depends on both prior and following words, common in conversational language.
- **Improving Response Coherence**: Since chatbots need to provide coherent responses, the contextual understanding offered by Bi-LSTMs leads to responses that are more relevant to the user's intent.

Practical Considerations for Bi-LSTM Chatbot Implementation

1. **Data Requirements**: A substantial and varied dataset of conversational pairs is necessary to train a Bi-LSTM-based chatbot effectively.
2. **Preprocessing**: Tokenization, removal of stopwords (optional), and converting words to embeddings are key steps in preparing data for the Bi-LSTM model.
3. **Hyperparameters**:
    - **Embedding size**: Determines the dimensionality of the word vectors.
    - **Hidden layer size**: Affects the model's capacity to capture context.
    - **Learning rate**: Controls the rate of model updates during training.
4. **Inference (Response Generation)**:
    - During inference, start the decoder with a special start-of-sequence token and generate words until an end-of-sequence token is produced or a max length is reached.

Advantages and Limitations

Advantages:

- **Contextual Understanding**: Bi-LSTMs improve understanding by capturing dependencies in both directions.
- **Adaptability**: Can generalize to various types of conversational data.

Limitations:

- **Resource Intensive**: Bi-LSTMs are computationally demanding, especially on longer sequences.
- **Training Data Needs**: Requires large conversational datasets to perform effectively.

Code:

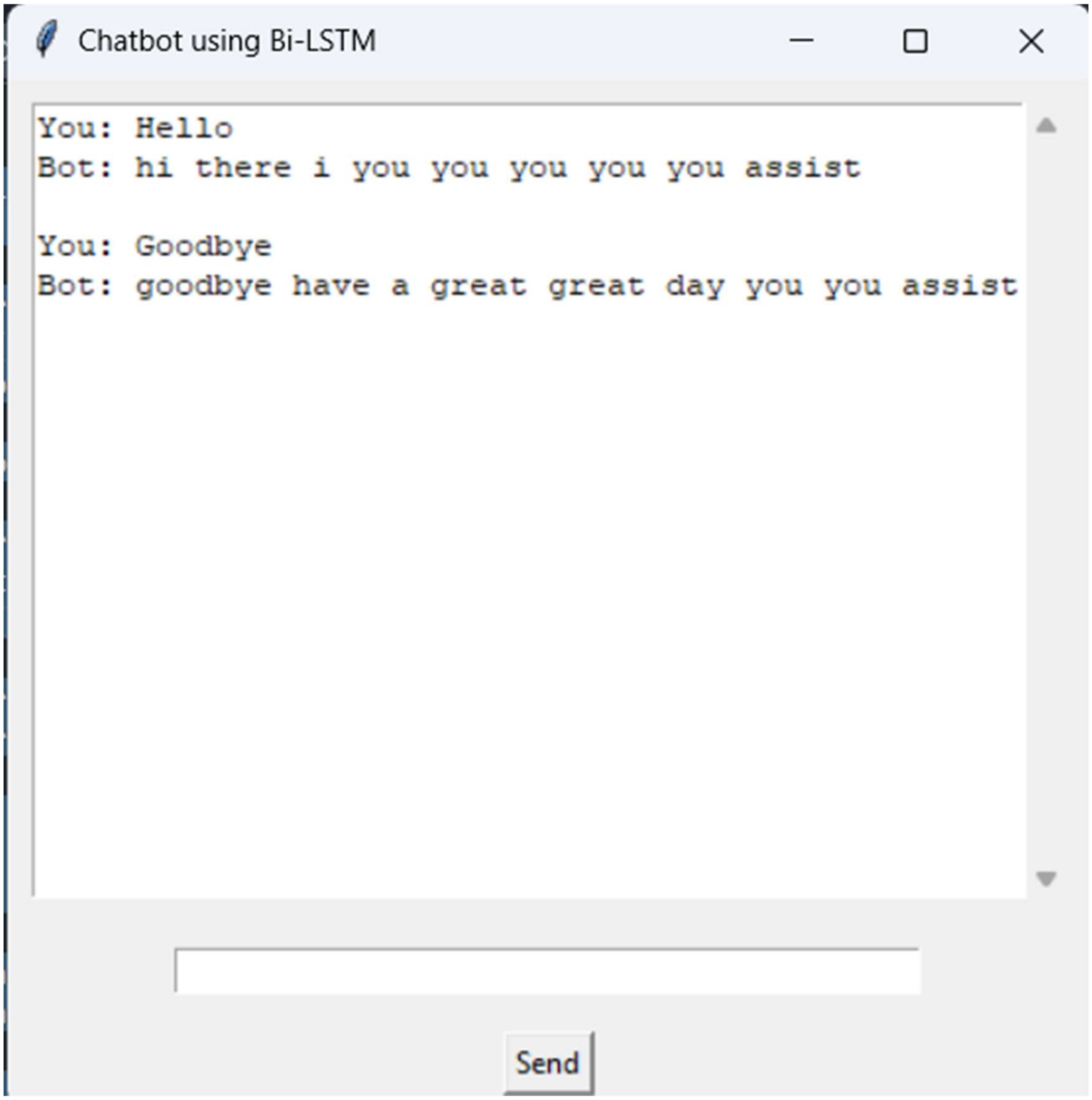
```
import tensorflow as tf
from tensorflow.keras.layers import Embedding, LSTM, Dense, Bidirectional
from tensorflow.keras.models import Sequential
import numpy as np
import nltk
from nltk.tokenize import word_tokenize
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import tkinter as tk
from tkinter import scrolledtext

# Download NLTK tokenizer data
nltk.download('punkt')

# Sample dataset (toy example for demonstration)
conversations = [
    ("Hello", "Hi there!"),
    ("How are you?", "I'm fine, thank you! How can I assist you?"),
    ("What is your name?", "I am a chatbot created for conversation."),
    ("Goodbye", "Goodbye! Have a great day!")
]

# Preprocess text data
input_texts, target_texts = zip(*conversations)
```

```
tokenizer = Tokenizer()
tokenizer.fit_on_texts(input_texts + target_texts)
input_sequences = tokenizer.texts_to_sequences(input_texts)
target_sequences = tokenizer.texts_to_sequences(target_texts)

# Padding sequences
max_len = max(len(seq) for seq in input_sequences + target_sequences)
input_sequences = pad_sequences(input_sequences, maxlen=max_len, padding='post')
target_sequences = pad_sequences(target_sequences, maxlen=max_len, padding='post')
vocab_size = len(tokenizer.word_index) + 1

# Prepare target data for training (shifted by one position)
target_data = np.zeros((len(target_sequences), max_len, vocab_size), dtype='float32')
for i, seq in enumerate(target_sequences):
    for t, word_id in enumerate(seq):
        if word_id != 0:
target_data[i, t, word_id] = 1.0

# Build the Bi-LSTM Model
model = Sequential([
    Embedding(input_dim=vocab_size, output_dim=128, input_length=max_len),
    Bidirectional(LSTM(256, return_sequences=True)),
    Dense(vocab_size, activation='softmax')
])

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

```
# Train the model
print("Training the model...")
model.fit(input_sequences, target_data, epochs=300, batch_size=8)
print("Training completed.")

# Define the prediction function
def predict_response(text):
input_sequence = tokenizer.texts_to_sequences([text])
input_sequence = pad_sequences(input_sequence, maxlen=max_len, padding='post')
    prediction = model.predict(input_sequence)
predicted_sequence = np.argmax(prediction[0], axis=1)

    response = []
    for word_id in predicted_sequence:
        if word_id == 0:
            continue
        word = tokenizer.index_word[word_id]
response.append(word)
    return ' '.join(response)

# Tkinter GUI for Chatbot
class ChatbotApp:
    def __init__(self, root):
self.root = root
self.root.title("Chatbot using Bi-LSTM")

self.chat_history = scrolledtext.ScrolledText(root, wrap=tk.WORD, width=50, height=20)
self.chat_history.pack(padx=10, pady=10)
```

```
self.user_input = tk.Entry(root, width=50)
self.user_input.pack(padx=10, pady=10)
self.user_input.bind("<Return>", self.get_response)

self.send_button = tk.Button(root, text="Send", command=self.get_response)
self.send_button.pack(padx=10, pady=5)

    def get_response(self, event=None):
user_text = self.user_input.get().strip()
        if user_text:
self.chat_history.insert(tk.END, "You: " + user_text + "\n")
            response = predict_response(user_text)
self.chat_history.insert(tk.END, "Bot: " + response + "\n\n")
self.chat_history.see(tk.END)  # Auto-scroll to the bottom
self.user_input.delete(0, tk.END)

# Initialize Tkinter app
root = tk.Tk()
app = ChatbotApp(root)
root.mainloop()
```

Output:


Chatbot using Bi-LSTM
You: Hello
Bot: hi there i you you you you you assist
You: Goodbye
Bot: goodbye have a great great day you you assist
Send

EXPERIMENT – 07

Aim: Image classification on MNIST dataset (CNN model with fully connected layer)

Theory:

For image classification on the MNIST dataset, we use a **Convolutional Neural Network (CNN)** to recognize handwritten digits (0-9). The MNIST dataset consists of 28x28 grayscale images of digits, with 60,000 training images and 10,000 test images.

CNN Architecture Overview

1. **Convolutional Layers**: These layers apply filters (kernels) to detect features like edges and textures. They capture spatial hierarchies in the image, helping the model recognize patterns specific to each digit.
2. **Activation (ReLU)**: The ReLU function introduces non-linearity, allowing the network to learn complex patterns.
3. **Pooling Layers**: Pooling reduces the spatial dimensions of the feature maps, which reduces computation and helps retain important features.
4. **Flatten Layer**: Converts the 2D feature maps into a 1D vector so it can be fed into the fully connected layers.
5. **Fully Connected Layers**: These dense layers combine features from the convolutional layers to make the final classification. The last layer uses **softmax** to output probabilities for each of the 10 digit classes.

Training

- **Loss Function**: Categorical Cross-Entropy is used to measure prediction accuracy.
- **Optimizer**: Optimizers like Adam adjust model weights to minimize the loss during training.

Result

The trained CNN can classify each image into one of the 10 digits, achieving high accuracy due to CNNs' ability to capture complex spatial features.

Code:

```
import torch

import torch.nn as nn

import torch.optim as optim

import torchvision

import torchvision.transforms as transforms
```

```
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

# Set random seed for reproducibility
torch.manual_seed(42)

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Hyperparameters
num_epochs = 10
batch_size = 64
learning_rate = 0.001

# MNIST dataset
transform = transforms.Compose([
transforms.ToTensor(),
transforms.Normalize((0.1307,), (0.3081,))
])

train_dataset = torchvision.datasets.MNIST(
    root='./data',
    train=True,
    transform=transform,
    download=True
)

test_dataset = torchvision.datasets.MNIST(
```

```
    root='./data',
    train=False,
    transform=transform,
    download=True
)

train_loader = DataLoader(
    dataset=train_dataset,
batch_size=batch_size,
    shuffle=True
)

test_loader = DataLoader(
    dataset=test_dataset,
batch_size=batch_size,
    shuffle=False
)

# CNN Model
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=2),
nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.conv2 = nn.Sequential(
```

```
            nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=2),
nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
self.fc = nn.Linear(7*7*32, 10)

    def forward(self, x):
        out = self.conv1(x)
        out = self.conv2(out)
        out = out.reshape(out.size(0), -1)
        out = self.fc(out)
        return out

# Initialize the model
model = ConvNet().to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Training function
def train_model():
model.train()
train_losses = []

    for epoch in range(num_epochs):
running_loss = 0.0
        for i, (images, labels) in enumerate(train_loader):
```

```
            images = images.to(device)
            labels = labels.to(device)

            # Forward pass
            outputs = model(images)
            loss = criterion(outputs, labels)

            # Backward and optimize
optimizer.zero_grad()
loss.backward()
optimizer.step()

running_loss += loss.item()

            if (i+1) % 100 == 0:
                print(f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{len(train_loader)}], Loss:
{loss.item():.4f}')

epoch_loss = running_loss / len(train_loader)
train_losses.append(epoch_loss)
        print(f'Epoch [{epoch+1}/{num_epochs}] Loss: {epoch_loss:.4f}')

    return train_losses

# Testing function
def test_model():
model.eval()
    with torch.no_grad():
        correct = 0
```

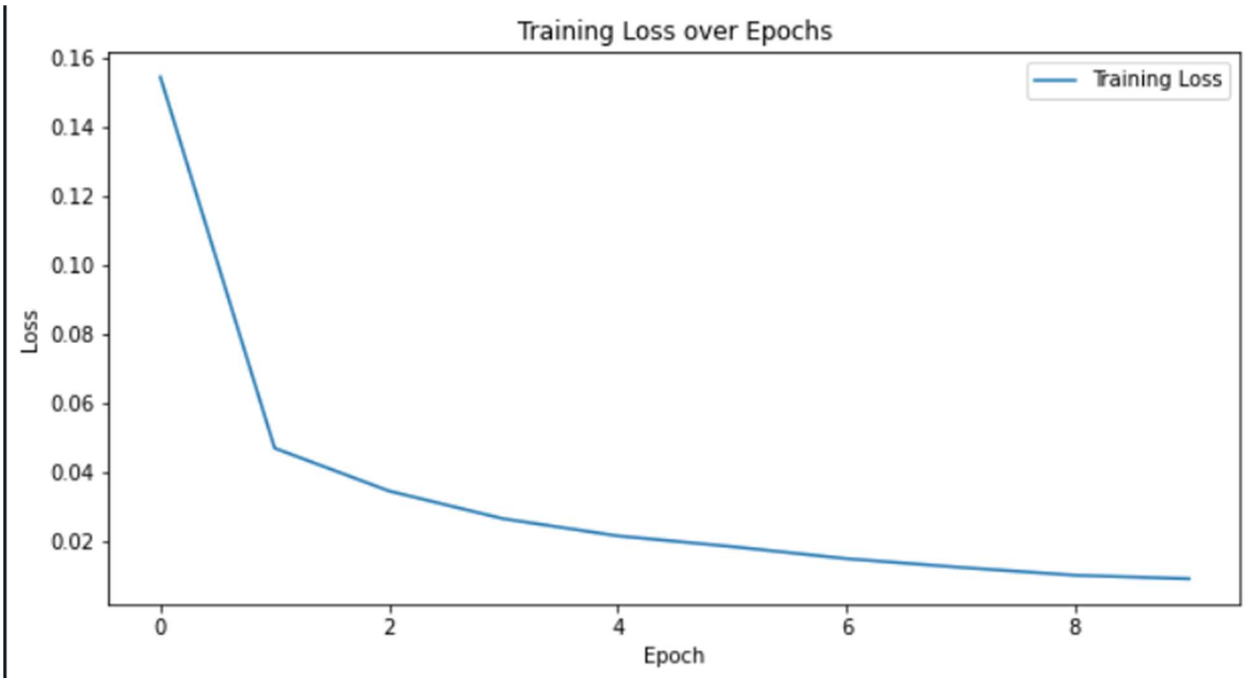
```
        total = 0
        for images, labels in test_loader:
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        accuracy = 100 * correct / total
        print(f'Test Accuracy: {accuracy:.2f}%')
        return accuracy

# Training the model
print("Starting training...")
train_losses = train_model()

# Testing the model
print("\nTesting the model...")
test_accuracy = test_model()

# Plotting training loss
plt.figure(figsize=(10,5))
plt.plot(train_losses, label='Training Loss')
plt.title('Training Loss over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
```

plt.show()

Save the model

torch.save(model.state_dict(), 'mnist_cnn.pth')

Output:




```
Testing the model...
Test Accuracy: 99.03%
```

EXPERIMENT – 08

Aim: Train a sentiment analysis model on IMDB dataset, use RNN layers with LSTM/GRU

Theory:

For training a sentiment analysis model on the IMDB dataset using Recurrent Neural Network (RNN) layers with **LSTM** (Long Short-Term Memory) or **GRU** (Gated Recurrent Unit), we analyze and predict the sentiment (positive or negative) of movie reviews. This is a supervised text classification task using sequential modeling.

IMDB Dataset Overview

The **IMDB dataset** contains 50,000 labeled movie reviews: 25,000 for training and 25,000 for testing. Each review is labeled as either positive or negative, making this a binary classification problem.

Why Use RNNs (LSTM/GRU) for Sentiment Analysis?

Sentiment analysis requires understanding the context within a sequence of words. RNNs, particularly LSTM and GRU, are well-suited for this as they can capture dependencies in sequences over long distances, making them ideal for handling the temporal nature of text.

Model Architecture

1. **Embedding Layer**: Converts words into dense vector representations, capturing semantic relationships in word embeddings. This layer maps each word to a lower-dimensional space, making it easier for the model to learn patterns in text data.
2. **LSTM/GRU Layers**:
   - **LSTM**: LSTM layers use gates (input, forget, and output gates) to control the flow of information, effectively capturing long-term dependencies and addressing the vanishing gradient problem of standard RNNs.
   - **GRU**: GRUs simplify LSTMs by using only two gates (update and reset gates), making them computationally lighter while retaining the ability to handle long-term dependencies.
3. **Fully Connected (Dense) Layer**: After the RNN layers, a dense layer is used to output predictions based on the learned features from the LSTM/GRU.
4. **Output Layer (Sigmoid)**: The final layer applies a sigmoid activation function to produce probabilities for the binary classification (positive or negative sentiment).

Training Process

- **Loss Function**: Binary cross-entropy is used for the loss function to measure the difference between the predicted probabilities and the true labels.

- **Optimizer**: Adam or RMSprop optimizers are commonly used for efficient training and convergence.

Evaluation

After training, the model's performance is evaluated on the test set. The accuracy and other metrics indicate how well the model understands sentiment within unseen reviews. The sequential nature of LSTM/GRU layers helps the model grasp context and subtle cues that signal sentiment, improving predictive performance on IMDB data.

Code:

```
import tensorflow as tf

from tensorflow.keras.datasets import imdb

from tensorflow.keras.preprocessing.sequence import pad_sequences

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout

from tensorflow.keras.callbacks import EarlyStopping

import numpy as np


# Set random seed for reproducibility

tf.random.set_seed(42)

np.random.seed(42)


# Parameters

max_features = 10000  # Maximum number of words to keep

maxlen = 200         # Max length of each review

embedding_dims = 100  # Dimension of embedding space

batch_size = 32

epochs = 10


# Load IMDB dataset

print("Loading IMDB dataset...")

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

```
# Pad sequences to ensure uniform length
print("Preprocessing data...")
x_train = pad_sequences(x_train, maxlen=maxlen)
x_test = pad_sequences(x_test, maxlen=maxlen)

# Build the model
print("Building model...")
model = Sequential([
    # Embedding layer to convert word indices to dense vectors
    Embedding(max_features, embedding_dims, input_length=maxlen),

    # First LSTM layer with return sequences for stacking
    LSTM(64, return_sequences=True),
    Dropout(0.3),

    # Second LSTM layer
    LSTM(32),
    Dropout(0.3),

    # Dense layers for classification
    Dense(64, activation='relu'),
    Dropout(0.3),
    Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(
```

```
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['accuracy']
)

# Model summary
model.summary()

# Early stopping callback
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=3,
restore_best_weights=True
)

# Train the model
print("\nTraining model...")
history = model.fit(
x_train, y_train,
batch_size=batch_size,
    epochs=epochs,
validation_split=0.2,
    callbacks=[early_stopping],
    verbose=1
)

# Evaluate the model
print("\nEvaluating model...")
```

```
test_loss, test_accuracy = model.evaluate(x_test, y_test, verbose=0)
print(f"Test accuracy: {test_accuracy:.4f}")
print(f"Test loss: {test_loss:.4f}")

# Function to predict sentiment for new reviews
def predict_sentiment(text, word_index=imdb.get_word_index()):
    # Reverse word index to get words from indices
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])

    # Convert text to sequence of indices
    tokens = tf.keras.preprocessing.text.text_to_word_sequence(text)
    indices = []
    for word in tokens:
        if word in word_index and word_index[word] <max_features:
indices.append(word_index[word])
        else:
indices.append(2)  # Unknown token

    # Pad sequence
    padded = pad_sequences([indices], maxlen=maxlen)

    # Get prediction
    prediction = model.predict(padded)[0][0]
    return {
        'sentiment': 'Positive' if prediction > 0.5 else 'Negative',
        'confidence': float(prediction if prediction > 0.5 else 1 - prediction)
    }
```

```
# Example usage of prediction
sample_review = "This movie was fantastic! The acting and directing were amazing."
result = predict_sentiment(sample_review)
print(f"\nSample Review: {sample_review}")
print(f"Prediction: {result['sentiment']} (Confidence: {result['confidence']:.2%})")
```

Output:

```
Evaluating model...
Test accuracy: 0.8508
Test loss: 0.3518
1/1 [==============================] - 0s 493ms/step

Sample Review: This movie was fantastic! The acting and directing were amazing.
Prediction: Positive (Confidence: 59.75%)
```

EXPERIMENT – 09

Aim: Applying the Deep Learning Models in the field of Natural Language Processing

Theory:

Deep Learning models have transformed **Natural Language Processing (NLP)** by enabling machines to understand, interpret, and generate human language. Key models include:

1. **Word Embeddings**: Techniques like Word2Vec and GloVe convert words into dense vectors, capturing word meanings and relationships.
2. **RNNs (LSTM/GRU)**: Recurrent Neural Networks, especially LSTM and GRU, capture sequence dependencies, making them ideal for tasks like sentiment analysis and text generation.
3. **CNNs**: Used in text classification, CNNs detect local patterns and dependencies in text sequences.
4. **Transformers**: Transformers (e.g., BERT, GPT) use self-attention to capture relationships across a sentence, handling context better and processing sequences in parallel. They excel in tasks like translation, question answering, and text generation.

Code:

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
import pandas as pd


# Load IMDB dataset (sample data for this example)
def load_data():
    data = {
        'review': [
            "I loved this movie. It was fantastic!",
            "This was a terrible movie. I hated it.",
            "What a great film! I really enjoyed it.",
```

```
            "It was boring and too long.",
            "Absolutely wonderful! A must-see.",
            "Not good at all. Very disappointing.",
            "An excellent film with great performances.",
            "The plot was predictable and dull.",
            "A masterpiece! I would watch it again.",
            "It was okay, not great but not bad either."
        ],
        'sentiment': [1, 0, 1, 0, 1, 0, 1, 0, 1, 0]  # 1 for positive, 0 for negative
    }
    return pd.DataFrame(data)

# Custom Dataset Class
class TextDataset(Dataset):
    def __init__(self, reviews, labels):
self.reviews = reviews
self.labels = labels

    def __len__(self):
        return len(self.reviews)

    def __getitem__(self, idx):
        return self.reviews[idx], self.labels[idx]

# CNN Model Definition with Dropout
class TextCNN(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(TextCNN, self).__init__()
```

```
self.embedding = nn.Embedding(input_dim, 100)  # Embedding layer
        self.conv1 = nn.Conv2d(1, 100, (3, 100))  # Convolutional layer with kernel size (3, 
embedding_dim)
        self.fc1 = nn.Linear(100, 50)  # Fully connected layer
        self.fc2 = nn.Linear(50, output_dim)  # Output layer
self.dropout = nn.Dropout(0.5)  # Dropout layer for regularization

    def forward(self, x):
        x = self.embedding(x)  # Get embeddings
        x = x.unsqueeze(1)  # Add channel dimension
        x = torch.relu(self.conv1(x))  # Convolutional layer
        x = nn.MaxPool2d((x.size(2), 1))(x)  # Max pooling over the height dimension
        x = x.view(x.size(0), -1)  # Flatten the output
        x = self.dropout(x)  # Apply dropout
        x = torch.relu(self.fc1(x))  # Fully connected layer with ReLU activation
        return self.fc2(x)  # Output layer

# Function to predict sentiment of new reviews
def predict_sentiment(model, vectorizer, review):
model.eval()
review_vectorized = vectorizer.transform([review]).toarray()
review_tensor = torch.LongTensor(review_vectorized)

    with torch.no_grad():
        output = model(review_tensor)
        _, predicted = torch.max(output.data, 1)

    return predicted.item()
```

```
# Main Function
def main():
    # Load data
df = load_data()

    # Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(df['review'], df['sentiment'], test_size=0.2)

    # Vectorize text data using CountVectorizer
    vectorizer = CountVectorizer(max_features=5000)  # Limit to top 5000 words for better
performance
X_train_vectorized = vectorizer.fit_transform(X_train).toarray()
X_test_vectorized = vectorizer.transform(X_test).toarray()

    # Create datasets and dataloaders
train_dataset = TextDataset(torch.LongTensor(X_train_vectorized),
torch.LongTensor(y_train.values))
test_dataset = TextDataset(torch.LongTensor(X_test_vectorized),
torch.LongTensor(y_test.values))

train_loader = DataLoader(train_dataset, batch_size=4, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=4)

    # Initialize model, loss function and optimizer
input_dim = X_train_vectorized.shape[1]
output_dim = 2  # Binary classification (positive/negative)

    model = TextCNN(input_dim=input_dim, output_dim=output_dim)
```

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

    # Training Loop
num_epochs = 100
    for epoch in range(num_epochs):
model.train()
        for reviews, labels in train_loader:
optimizer.zero_grad()
            outputs = model(reviews)
            loss = criterion(outputs, labels)
loss.backward()
optimizer.step()

        if (epoch + 1) % 10 == 0:
            print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}')

    # Evaluation Loop
model.eval()
    correct = 0
    total = 0

predictions_list = []

    with torch.no_grad():
        for reviews, labels in test_loader:
            outputs = model(reviews)
            _, predicted = torch.max(outputs.data, 1)
```

```
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

predictions_list.extend(predicted.numpy())   # Collect predictions

    accuracy = f'Accuracy of the model on the test set: {100 * correct / total:.2f}%'

    print(accuracy)

    print("\nPredictions vs Actual Sentiments:")

    for idx in range(len(predictions_list)):
        print(f"Predicted: {predictions_list[idx]}, Actual: {y_test.iloc[idx]}")

    # Test the model with some sample inputs
sample_reviews = [
        "I absolutely loved this movie! It was amazing!",
        "This is the worst film I have ever seen.",
        "It was just okay; nothing special.",
        "An outstanding performance by the lead actor!",
        "I didn't like it at all."
    ]

    print("\nTesting Sample Inputs:")

    for review in sample_reviews:
sentiment_prediction = predict_sentiment(model, vectorizer, review)
sentiment_label = "Positive" if sentiment_prediction == 1 else "Negative"
```

```
        print(f"Review: \"{review}\" => Predicted Sentiment: {sentiment_label}")


if __name__ == "__main__":
    main()
```

Output:

```
Epoch [10/100], Loss: 0.6975
Epoch [20/100], Loss: 0.5816
Epoch [30/100], Loss: 0.4711
Epoch [40/100], Loss: 0.6915
Epoch [50/100], Loss: 0.5172
Epoch [60/100], Loss: 0.7380
Epoch [70/100], Loss: 0.5097
Epoch [80/100], Loss: 0.7530
Epoch [90/100], Loss: 0.5931
Epoch [100/100], Loss: 0.6646
Accuracy of the model on the test set: 50.00%

Predictions vs Actual Sentiments:
Predicted: 1, Actual: 1
Predicted: 1, Actual: 0
```

```
Testing Sample Inputs:
Review: "I absolutely loved this movie! It was amazing!" => Predicted Sentiment: Positive
Review: "This is the worst film I have ever seen." => Predicted Sentiment: Negative
Review: "It was just okay; nothing special." => Predicted Sentiment: Positive
Review: "An outstanding performance by the lead actor!" => Predicted Sentiment: Positive
Review: "I didn't like it at all." => Predicted Sentiment: Positive
```

EXPERIMENT – 10

Aim: Program to demonstrate K-Means Clustering Algorithm on Handwritten Dataset

Theory:

K-Means Clustering is a popular unsupervised machine learning algorithm used for partitioning a dataset into distinct groups, or clusters, based on feature similarities. The primary objective of K-Means is to divide nnn observations into kkk clusters in which each observation belongs to the cluster with the nearest mean (centroid), minimizing the overall within-cluster variance.

Key Concepts

1. **Initialization**:
    - The algorithm starts by selecting kkk initial centroids, which can be chosen randomly from the dataset or by using methods like K-Means++ for better initial placement.
2. **Assignment Step**:
    - Each data point is assigned to the nearest centroid, forming kkk clusters. The distance between data points and centroids is typically calculated using Euclidean distance.
3. **Update Step**:
    - After assigning all data points, the centroids of the clusters are recalculated by taking the mean of all points assigned to each cluster.
4. **Iteration**:
    - Steps 2 and 3 are repeated until convergence, which occurs when the centroids no longer change significantly or when a predetermined number of iterations is reached.
5. **Output**:
    - The algorithm outputs the final centroids and the cluster assignments for each data point.

Advantages of K-Means

- **Simplicity**: Easy to understand and implement.
- **Efficiency**: Fast convergence on small to medium-sized datasets.
- **Scalability**: Can handle large datasets effectively.

Limitations of K-Means

- **Choice of kkk**: The number of clusters kkk must be specified beforehand, which may not be intuitive.
- **Sensitivity to Initialization**: Poor initialization can lead to suboptimal solutions.
- **Shape of Clusters**: K-Means assumes spherical clusters of similar sizes, which may not be appropriate for all datasets.
- **Outliers**: Sensitive to outliers, as they can skew the centroids.

Application: K-Means on Handwritten Dataset

In the context of handwritten digit recognition (like the MNIST dataset), K-Means clustering can be used to group similar handwritten digits based on their pixel values. Each digit image can be represented as a high-dimensional vector, and K-Means can identify clusters that represent different digit classes.

Steps to Demonstrate K-Means on Handwritten Dataset

1. **Load the Dataset**: Import the handwritten dataset (e.g., MNIST).
2. **Preprocess the Data**: Normalize the pixel values and flatten the images into vectors.
3. **Choose kkk**: Select the number of clusters (usually 10 for digits 0-9).
4. **Run K-Means**: Apply the K-Means algorithm to cluster the data.
5. **Visualize Results**: Display the cluster centers and some data points from each cluster to assess the effectiveness of the clustering.

Code :

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import numpy as np

# Hyperparameters
num_epochs = 10
```

```
batch_size = 4
learning_rate = 0.001

    # Transformations for the training and testing data
    transform = transforms.Compose([
transforms.ToTensor(),
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),  # Normalize to [-1, 1]
    ])

    # Load CIFAR-10 dataset
    trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                          download=True, transform=transform)
trainloader = DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                          download=True, transform=transform)
testloader = DataLoader(testset, batch_size=batch_size,
                                          shuffle=False)

    # Define CNN Model
    class CNN(nn.Module):
        def __init__(self):
            super(CNN, self).__init__()
            self.conv1 = nn.Conv2d(3, 6, 5)   # Input: 3 channels (RGB), Output: 6 channels
self.pool = nn.MaxPool2d(2, 2)      # Max pooling layer
            self.conv2 = nn.Conv2d(6, 16, 5)   # Input: 6 channels, Output: 16 channels
            self.fc1 = nn.Linear(16 * 5 * 5, 120) # Fully connected layer
```

```
        self.fc2 = nn.Linear(120, 84)      # Fully connected layer
        self.fc3 = nn.Linear(84, 10)       # Output layer for 10 classes

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x))) # Convolution + ReLU + Pooling
        x = self.pool(torch.relu(self.conv2(x))) # Convolution + ReLU + Pooling
        x = x.view(-1, 16 * 5 * 5)             # Flatten the output
        x = torch.relu(self.fc1(x))            # Fully connected layer + ReLU
        x = torch.relu(self.fc2(x))            # Fully connected layer + ReLU
        return self.fc3(x)                     # Output layer

# Initialize model, loss function and optimizer
model = CNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Training Loop
for epoch in range(num_epochs):
    for inputs, labels in trainloader:
optimizer.zero_grad()               # Zero the gradients
        outputs = model(inputs)             # Forward pass
        loss = criterion(outputs, labels)    # Compute loss
loss.backward()                     # Backward pass
optimizer.step()                    # Update weights

    print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}')

# Testing the Model on Test Data
```

```
model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, labels in testloader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print(f'Accuracy of the model on the test set: {100 * correct / total:.2f}%')

    # Test the model with some sample inputs from the test set and visualize them.
    def imshow(img):
img = img / 2 + 0.5     # Unnormalize the image
npimg = img.numpy()
plt.imshow(np.transpose(npimg, (1, 2, 0)))
plt.show()

dataiter = iter(testloader)
    images, labels = next(dataiter)

    # Print images and their predicted labels.
    outputs = model(images)
    _, predicted_labels = torch.max(outputs.data, 1)

    print("Predicted Labels:", predicted_labels.numpy())
```

```
print("Actual Labels:", labels.numpy())

# Show images with predicted labels.
imshow(torchvision.utils.make_grid(images))
```

Output:

```
Files already downloaded and verified
Files already downloaded and verified
Epoch [1/10], Loss: 1.6275
Epoch [2/10], Loss: 2.0582
Epoch [3/10], Loss: 0.8778
Epoch [4/10], Loss: 0.3380
Epoch [5/10], Loss: 0.3047
Epoch [6/10], Loss: 1.1177
Epoch [7/10], Loss: 0.8157
Epoch [8/10], Loss: 0.3056
Epoch [9/10], Loss: 0.5922
Epoch [10/10], Loss: 1.4003
Accuracy of the model on the test set: 60.49%
Predicted Labels: [3 8 9 0]
Actual Labels: [3 8 8 0]
```