***REINFORCEMENT LEARNING AND DEEP LEARNING LAB***

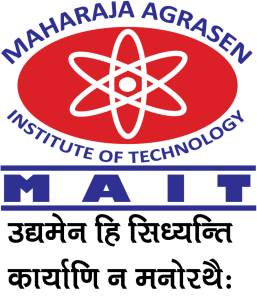
***ML-409P***

**Faculty Name:** Prof. (Dr.) Sachin Gupta **Student Name:** Yashi Mittal

**Roll No.:** 35314802721

**Semester:** 7th

**Group:** CSE (AIML-2B)

****

Maharaja Agrasen Institute of Technology, PSP Area,

Sector – 22, Rohini, New Delhi - 110085

**MACHINE LEARNING LAB:PRACTICAL RECORD**

**PAPER CODE : ML-409P**

Name of the student : Yashi Mittal

University Roll No. : 35314802721

Branch : CSE

Section/ Group : AIML-2B

**PRACTICAL DETAILS**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Exp**  **No.** | **Pg No.** | **Date** | **Experiment Name** | **Marks (0-3)** | | | | | **Total Marks (15)** | **Signature** |
|  |  |  |  | **R1** | **R2** | **R3** | **R4** | **R5** |  |  |
| 1. | 04 |  | Setting up the Spyder IDE Environment and Executing a Python Program |  |  |  |  |  |  |  |
| 2. | 06 |  | Installing Keras, Tensorflow and Pytorch libraries and making use of them |  |  |  |  |  |  |  |
| 3. | 11 |  | Implement Qlearning with pure Python to play a game  •Environment set up and intro to OpenAI Gym  •Write Qlearning algorithm and train agent to play game  • Watch trained agent play game |  |  |  |  |  |  |  |
| 4. | 17 |  | Implement deep Qnetwork with PyTorch |  |  |  |  |  |  |  |
| 5. | 31 |  | Python implementation of the iterative policy evaluation and update. |  |  |  |  |  |  |  |
| 6. | 40 |  | Chatbot using bi‐directional LSTMs |  |  |  |  |  |  |  |
| 7. | 48 |  | Image classification on MNIST dataset (CNN model with fully connected layer) |  |  |  |  |  |  |  |
| 8. | 55 |  | Train a sentiment analysis model on IMDB dataset, use RNN layers with LSTM/GRU |  |  |  |  |  |  |  |
| 9. | 61 |  | Applying the Deep Learning Models in the field of Natural Language Processing |  |  |  |  |  |  |  |
| 10. | 68 |  | Applying the Convolution Neural Network on computer vision problems |  |  |  |  |  |  |  |

**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:**

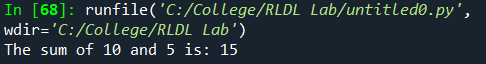
num1 = 10

num2 = 5

sum\_result = num1 + num2

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

**Output:**

****

**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

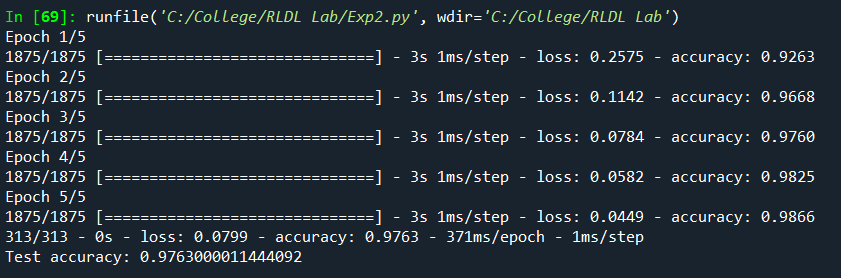
model.fit(x\_train, y\_train, epochs=5)

# Evaluate the model

test\_loss, test\_acc = model.evaluate(x\_test, y\_test, verbose=2)

print(f'Test accuracy: {test\_acc}')

**Output 1:**

****

**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)

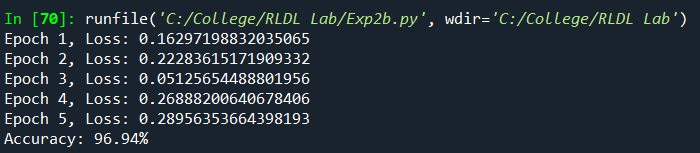
\_, predicted = torch.max(outputs.data, 1)

total += labels.size(0)

correct += (predicted == labels).sum().item()

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

**Output 2:**

****

**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′max​Q(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()

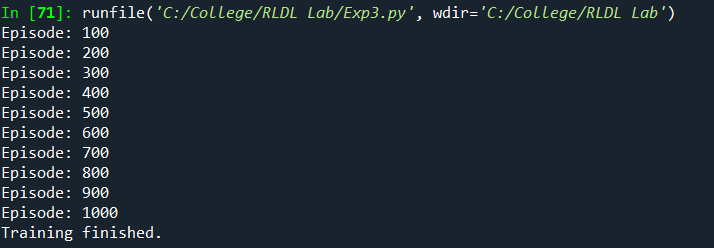
# Train the agent using Q-learning

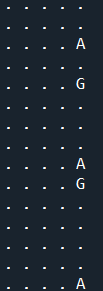
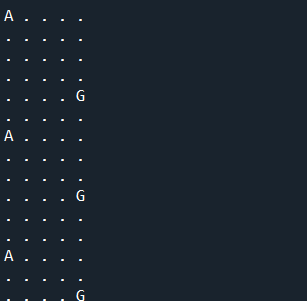
Q\_table = q\_learning(env, episodes=1000)

# Watch the trained agent

watch\_agent(env, Q\_table)

**Output**:



**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′max​Q(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...")

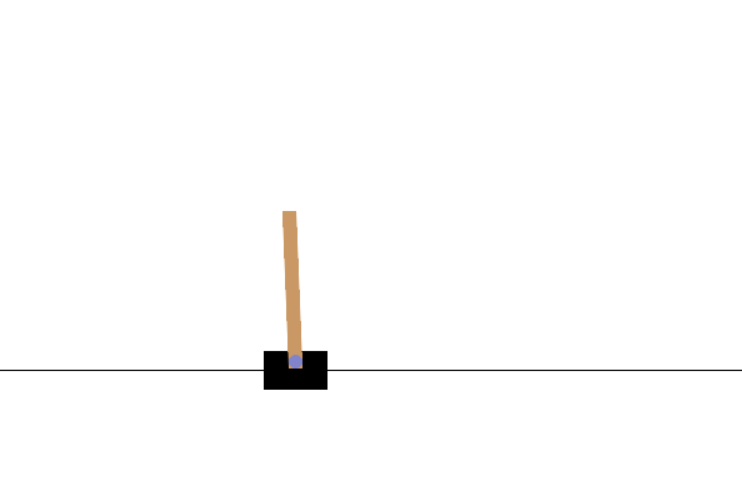
test\_agent(env, agent, episodes=5, render=True)

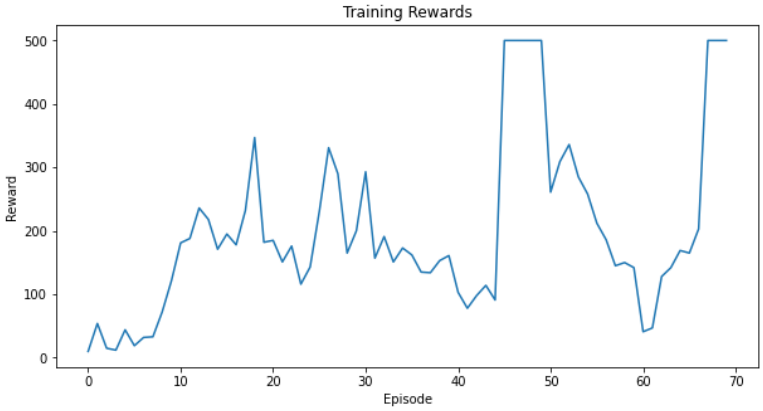
# Save the trained model

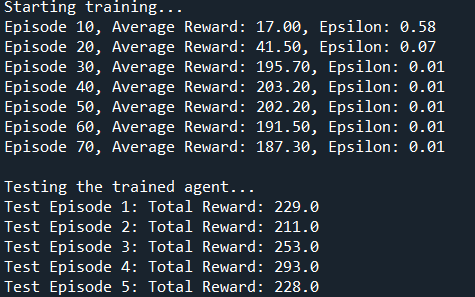
torch.save(agent.policy\_net.state\_dict(), 'dqn\_cartpole.pth')

print("\nModel saved to 'dqn\_cartpole.pth'")

**Output:**

****





**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))

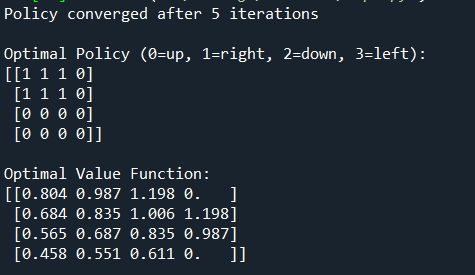
print("\nOptimal Value Function:")

print(np.round(optimal\_values.reshape(size, size), 3))

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Output:**

****

**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)

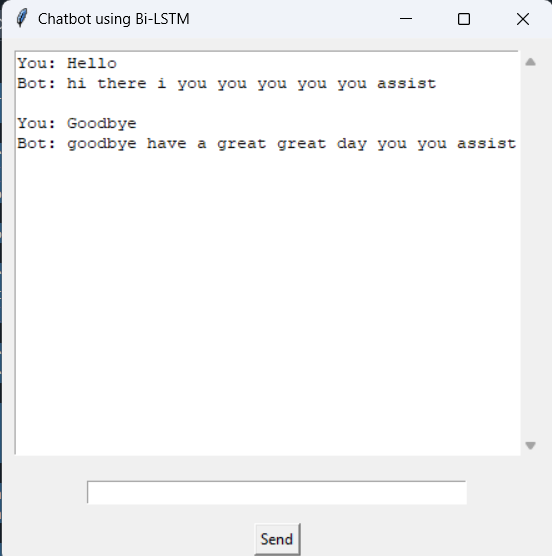
# Initialize Tkinter app

root = tk.Tk()

app = ChatbotApp(root)

root.mainloop()

**Output:**

****

**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')

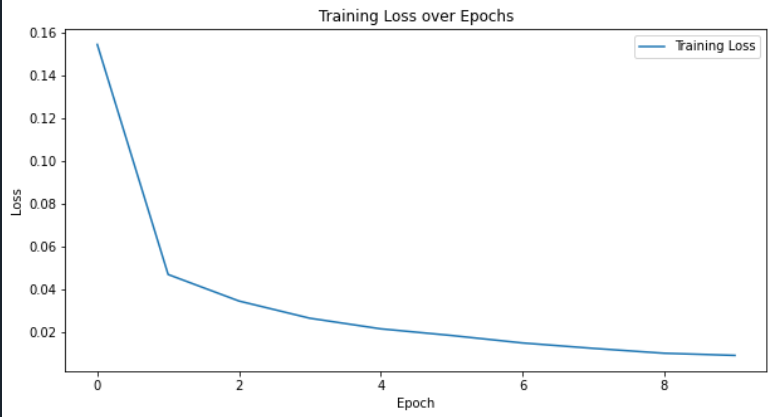
plt.legend()

plt.show()

# Save the model

torch.save(model.state\_dict(), 'mnist\_cnn.pth')

**Output:**





**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

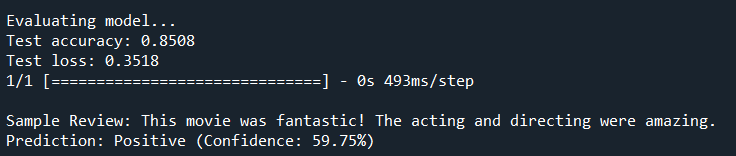
sample\_review = "This movie was fantastic! The acting and directing were amazing."

result = predict\_sentiment(sample\_review)

print(f"\nSample Review: {sample\_review}")

print(f"Prediction: {result['sentiment']} (Confidence: {result['confidence']:.2%})")

**Output:**



**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)

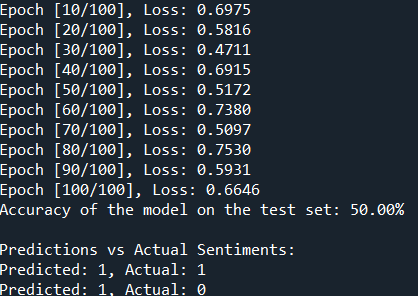
sentiment\_label = "Positive" if sentiment\_prediction == 1 else "Negative"

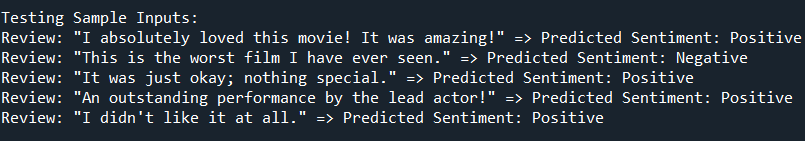
print(f"Review: \"{review}\" => Predicted Sentiment: {sentiment\_label}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Output:**

****



**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**:

