Reinforcement Learning and Deep Learning Lab

Faculty Name: Mr. Ajay Tiwari

Student Name: Prathya Thakur

Roll No.: 03614802721

Semester: 7th

Group: AIML 1 B

Maharaja Agrasen Institute of Technology, PSP Area, Sector - 22, New Delhi – 110085

MAHARAJA AGRASEN INSTITUTE OF TECHNOLOGY

COMPUTER SCIENCE & ENGINEERING DEPARTMENT

VISION

"To be centre of excellence in education, research and technology transfer in the field of computer engineering and promote entrepreneurship and ethical values."

MISSION

"To foster an open, multidisciplinary and highly collaborative research environment to produce world-class engineers capable of providing innovative solutions to real life problems and fulfill societal needs."

PRACTICAL RECORD

Name of the student : Prathya Thakur

University Roll No. : 03614802721

Branch : CSE-1

Section/ Group : 7 AIML 1 B

Exp. No.	Experiment Name	Date of performance	Date of checking	R1 (3)	R2 (3)	R3 (3)	R4 (3)	R5 (3)	Total Marks (15)	Signature
1.										
2.										
3.										
4.										
5.										
6.										
7.										
8.										
9.										

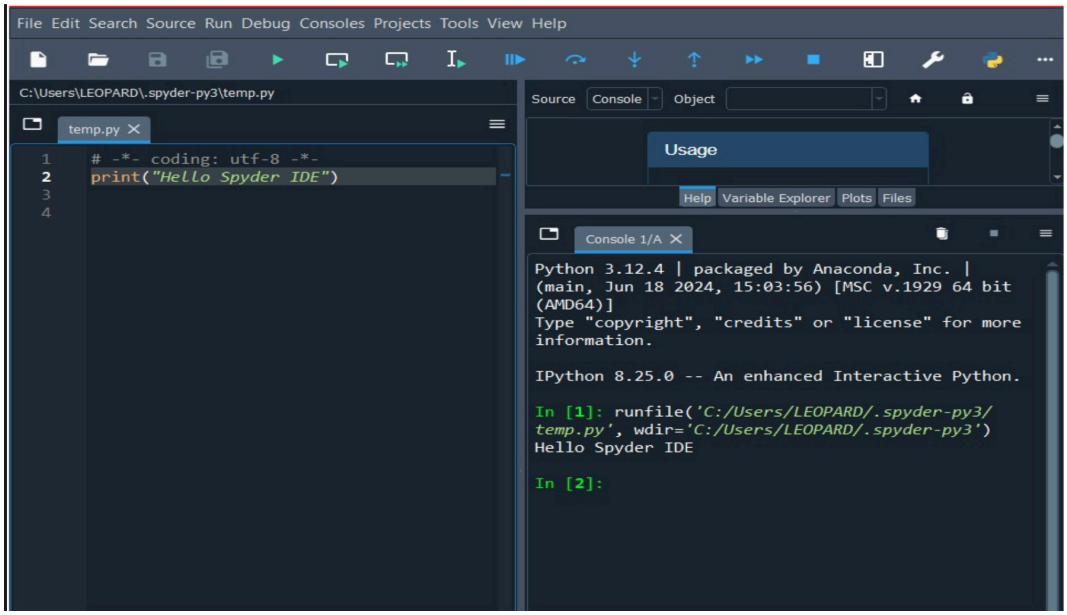
Exp. No.	Experiment Name	Date of performance	Date of checking	R1 (3)	R2 (3)	R3 (3)	R4 (3)	R5 (3)	Total Marks (15)	Signature
10.										
11.										
12.										
13.										
14.										
15.										

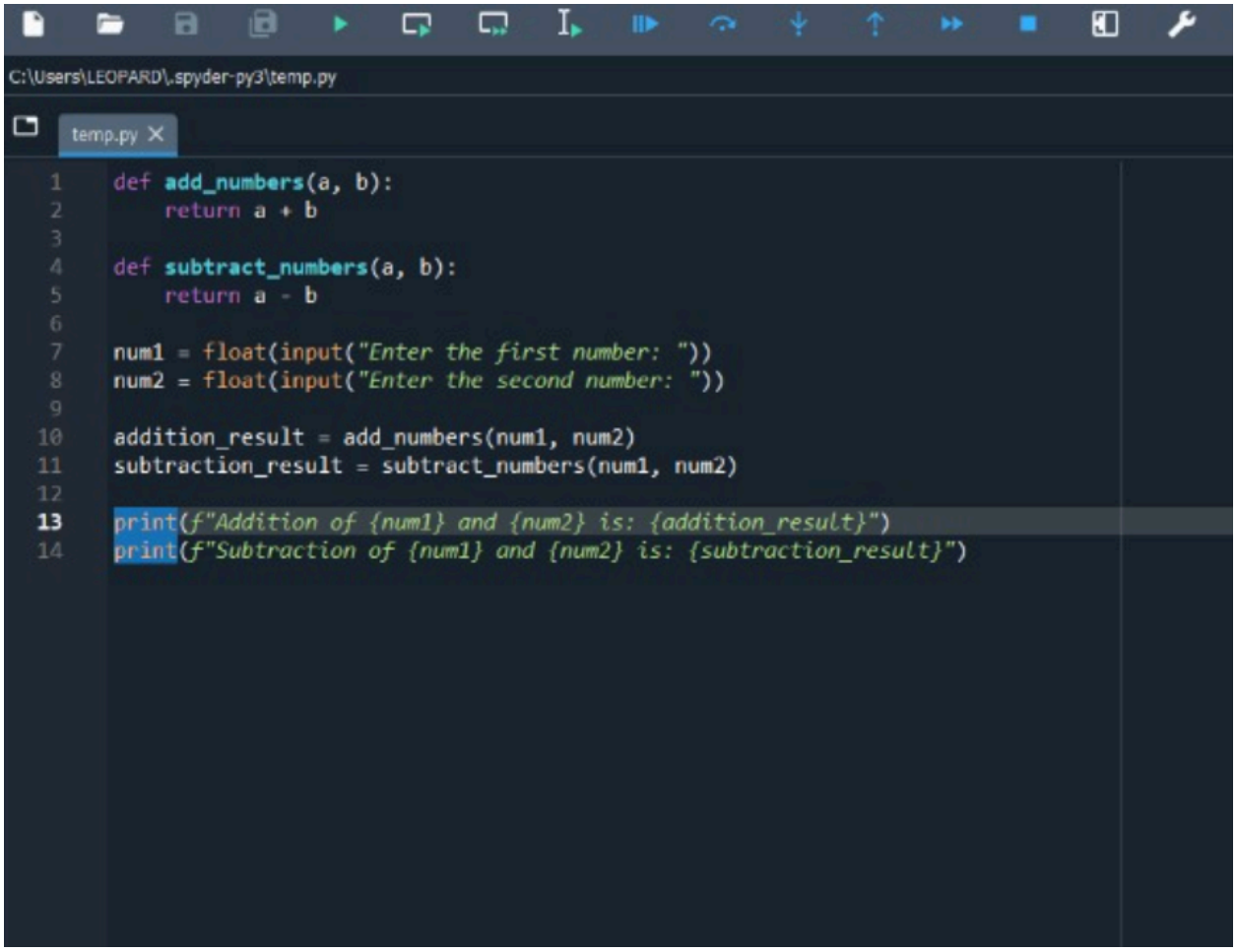
EXPERIMENT 1

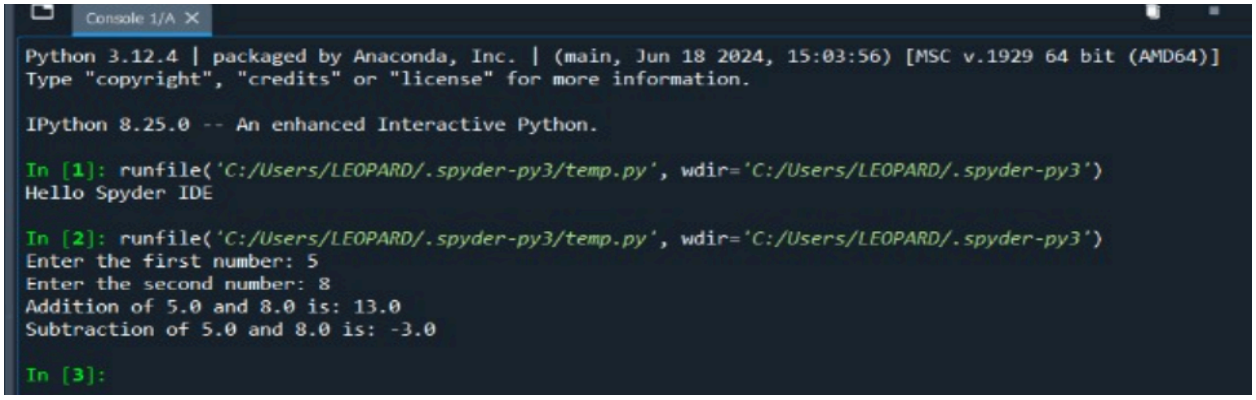
Aim: To set up the Spyder IDE for executing Python programs and running a basic Python script.

Theory: Spyder is an open-source integrated development environment (IDE) designed for Python, often used in data science. It allows users to write, debug, and execute Python code. The environment supports powerful libraries and provides a user-friendly interface for beginners and experts alike.

Output:




C:\Users\LEOPARD\.spyder-py3\temp.py
temp.py
1 def add_numbers(a, b):
2 return a + b
3
4 def subtract_numbers(a, b):
5 return a - b
6
7 num1 = float(input("Enter the first number: "))
8 num2 = float(input("Enter the second number: "))
9
10 addition_result = add_numbers(num1, num2)
11 subtraction_result = subtract_numbers(num1, num2)
12
13 print(f"Addition of {num1} and {num2} is: {addition_result}")
14 print(f"Subtraction of {num1} and {num2} is: {subtraction_result}")


Console 1/A
Python 3.12.4 | packaged by Anaconda, Inc. | (main, Jun 18 2024, 15:03:56) [MSC v.1929 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.
IPython 8.25.0 -- An enhanced Interactive Python.
In [1]: runfile('C:/Users/LEOPARD/.spyder-py3/temp.py', wdir='C:/Users/LEOPARD/.spyder-py3')
Hello Spyder IDE
In [2]: runfile('C:/Users/LEOPARD/.spyder-py3/temp.py', wdir='C:/Users/LEOPARD/.spyder-py3')
Enter the first number: 5
Enter the second number: 8
Addition of 5.0 and 8.0 is: 13.0
Subtraction of 5.0 and 8.0 is: -3.0
In [3]:

EXPERIMENT 2

Aim: To install Keras, TensorFlow, and PyTorch libraries and demonstrate their use in machine learning projects.

Theory: Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow. TensorFlow is an open-source platform for machine learning, while PyTorch is an open-source machine learning library based on Torch, primarily used for applications such as natural language processing and deep learning.

Source Code:

```
pip install tensorflow
```

```
Requirement already satisfied: tensorflow in
Requirement already satisfied: absl-py>=1.0.0
```

```
pip install torch
```

```
Collecting torch
  Downloading torch-2.2.2-cp312-none-macosx_1
Requirement already satisfied: filelock in /L
13.1)
Requirement already satisfied: typing-extensi
(from torch) (4.10.0)
```

```
pip install keras
```

```
Requirement already satisfied: keras in /Libr
Requirement already satisfied: absl-py in /Li
1.0)
```

IMPLEMENTATION CODE:

```
import tensorflow as tf
import torch
from tensorflow import keras

# TensorFlow model
model_tf = tf.keras.Sequential([tf.keras.layers.Dense(10, activation='relu'), tf.keras.layers.Dense(1)])
model_tf.compile(optimizer='adam', loss='mean_squared_error')
output_tf = model_tf(tf.random.normal([1, 5]))  # Random input for TensorFlow model

# PyTorch tensor and simple operation
x_pt = torch.tensor([1.0, 2.0, 3.0])
output_pt = torch.sum(x_pt ** 2)  # Squaring and summing PyTorch tensor elements

# Keras model
model_keras = keras.Sequential([keras.layers.Dense(10, activation='relu'), keras.layers.Dense(1)])
model_keras.compile(optimizer='adam', loss='mean_squared_error')
output_keras = model_keras.predict(tf.random.normal([1, 5]))  # Random input for Keras model

# Display outputs
print("TensorFlow Model Output:", output_tf)
print("PyTorch Tensor Sum:", output_pt.item())
print("Keras Model Output:", output_keras)
```

Output:

```
import keras
import tensorflow as tf
import torch

print(f"Keras Version: {keras.__version__}")
print(f"TensorFlow Version: {tf.__version__}")
print(f"PyTorch Version: {torch.__version__}")
```

```
2024-09-11 20:20:18.478879: I tensorflow/core/pl
nstructions in performance-critical operations.
To enable the following instructions: AVX2 FMA,
Keras Version: 3.0.5
TensorFlow Version: 2.16.1
PyTorch Version: 2.2.2
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 270ms/step
TensorFlow Model Output: tf.Tensor([[-0.21675241]], shape=(1, 1), dtype=float32)
PyTorch Tensor Sum: 14.0
Keras Model Output: [[-0.02438454]]
```

EXPERIMENT 3

Aim: To implement the Q-learning algorithm in pure Python and train the agent to play a game using the OpenAI Gym environment.

Theory: Q-learning is a model-free reinforcement learning algorithm that seeks to find the best action to take given the current state. It updates a Q-table that stores the cumulative expected reward for each action in each state. OpenAI Gym provides a collection of environments where agents can be trained and tested.

Source Code:

From the code and context, the game being played is FrozenLake-v1 from OpenAI Gym, specifically with is_slippery=False. This environment simulates an agent trying to navigate across a grid of frozen ground and holes (pits) to reach a goal.

Environment Setup and Introduction to OpenAI Gym:

```
import gym
import numpy as np

# Define Q-learning parameters
env = gym.make('CartPole-v1')  # Replace 'CartPole-v1' with your desired environment
num_episodes = 1000
learning_rate = 0.1
discount_factor = 0.99
epsilon = 1.0
max_epsilon = 1.0
min_epsilon = 0.01
decay_rate = 0.01
```

Write Q learning algo and train the Agent to play the game

```
# Initialize Q-table
num_states = env.observation_space.shape[0]
num_actions = env.action_space.n
Q_table = np.zeros((num_states, num_actions))

# Q-learning algorithm
for episode in range(num_episodes):
    state = env.reset()
    done = False
    truncated = False
    total_reward = 0

    while not done and not truncated:
        # Choose action using epsilon-greedy policy
        if np.random.rand() < epsilon:
            action = np.random.randint(0, num_actions)
        else:
            action = np.argmax(Q_table[state])

        # Take action and observe reward and next state
        next_state, reward, done, truncated, _ = env.step(action)

        total_reward += reward

        # Extract the first element of next_state if it's an array
        if isinstance(next_state, np.ndarray):
            next_state = next_state[0]

        # Ensure state and action are integers
        state = int(state)
        action = int(action)

        # Update Q-table using Q-learning update rule
        target = reward + discount_factor * np.max(Q_table[next_state])
```

```
            next_state = next_state[0]

        # Ensure state and action are integers
        state = int(state)
        action = int(action)

        # Update Q-table using Q-learning update rule
        target = reward + discount_factor * np.max(Q_table[next_state])
        Q_table[state][action] = (1 - learning_rate) * Q_table[state][action] + learning

        # Update state
        state = next_state

        # Decay epsilon
        epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.exp(-decay_rate * episo

    print(f"Episode {episode}: Total Reward = {total_reward}")
```

Watch Trained Agent play the game

```
# Evaluate the trained agent
num_eval_episodes = 10
for episode in range(num_eval_episodes):
    state = env.reset()
    done = False
    truncated = False
    total_reward = 0

    while not done and not truncated:
        action = np.argmax(Q_table[state])
        next_state, reward, done, truncated, _ = env.step(action)
        total_reward += reward
        state = next_state

    print(f"Evaluation Episode {episode}: Total Reward = {total_reward}")
```

```
num_eval_episodes = 10

for episode in range(num_eval_episodes):
    # In newer Gym versions, env.reset() returns (state, info)
    state, _ = env.reset()  # Unpack the tuple returned by reset
    done = False
    total_reward = 0

    while not done:
        # Ensure state is an integer
        state = int(state)

        # Choose action based on learned Q-table (greedy policy)
        action = np.argmax(Q_table[state])

        # Take action and observe the next state and reward
        next_state, reward, done, truncated, _ = env.step(action)

        # Add the reward to the total reward
        total_reward += reward

        # Update state
        state = next_state

    print(f"Evaluation Episode {episode}: Total Reward = {total_reward}")
```

Output:

```
Episode 0: Total Reward = 0.0
Episode 1: Total Reward = 0.0
Episode 2: Total Reward = 0.0
Episode 3: Total Reward = 0.0
Episode 4: Total Reward = 0.0
Episode 5: Total Reward = 0.0
Episode 6: Total Reward = 0.0
Episode 7: Total Reward = 0.0
Episode 8: Total Reward = 0.0
Episode 9: Total Reward = 0.0
Episode 10: Total Reward = 0.0
Episode 11: Total Reward = 0.0
Episode 12: Total Reward = 0.0
Episode 13: Total Reward = 0.0
Episode 14: Total Reward = 0.0
Episode 15: Total Reward = 0.0
Episode 16: Total Reward = 0.0
Episode 17: Total Reward = 0.0
```

WATCH THE MODEL PLAY:

```
Evaluation Episode 0: Total Reward = 1.0
Evaluation Episode 1: Total Reward = 1.0
Evaluation Episode 2: Total Reward = 1.0
Evaluation Episode 3: Total Reward = 1.0
Evaluation Episode 4: Total Reward = 1.0
Evaluation Episode 5: Total Reward = 1.0
Evaluation Episode 6: Total Reward = 1.0
Evaluation Episode 7: Total Reward = 1.0
Evaluation Episode 8: Total Reward = 1.0
Evaluation Episode 9: Total Reward = 1.0
```

EXPERIMENT 4

Aim: To implement a Deep Q-Network (DQN) using PyTorch for reinforcement learning tasks.

Theory: Deep Q-Network is a reinforcement learning algorithm that uses a deep neural network to approximate the optimal action-value function. It is commonly used in environments where an agent interacts with a dynamic environment to maximize cumulative rewards.

Source Code:

```
import gymnasium as gym
import math
import random
import matplotlib
import matplotlib.pyplot as plt
from collections import namedtuple, deque
from itertools import count

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

env = gym.make("CartPole-v1")

# set up matplotlib
is_ipython = 'inline' in matplotlib.get_backend()
if is_ipython:
    from IPython import display

plt.ion()

# if gpu is to be used
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
Transition = namedtuple('Transition', ('state', 'action', 'next_state', 'reward'))


class ReplayMemory(object):

    def __init__(self, capacity):
        self.memory = deque([], maxlen=capacity)

    def push(self, *args):
        """Save a transition"""
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

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

```python
class DQN(nn.Module):

    def __init__(self, n_observations, n_actions):
        super(DQN, self).__init__()
        self.layer1 = nn.Linear(n_observations, 128)
        self.layer2 = nn.Linear(128, 128)
        self.layer3 = nn.Linear(128, n_actions)

    # Called with either one element to determine next action, or a batch
    # during optimization. Returns tensor([[left0exp,right0exp]...]).
    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        return self.layer3(x)
```

```python
# BATCH_SIZE is the number of transitions sampled from the replay buffer
# GAMMA is the discount factor as mentioned in the previous section
# EPS_START is the starting value of epsilon
# EPS_END is the final value of epsilon
# EPS_DECAY controls the rate of exponential decay of epsilon, higher means a slower decay
# TAU is the update rate of the target network
# LR is the learning rate of the AdamW optimizer
BATCH_SIZE = 128
GAMMA = 0.99
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 1000
TAU = 0.005
LR = 1e-4

# Get number of actions from gym action space
n_actions = env.action_space.n
# Get the number of state observations
state, info = env.reset()
n_observations = len(state)

policy_net = DQN(n_observations, n_actions).to(device)
target_net = DQN(n_observations, n_actions).to(device)
target_net.load_state_dict(policy_net.state_dict())

optimizer = optim.AdamW(policy_net.parameters(), lr=LR, amsgrad=True)
memory = ReplayMemory(10000)
```

```python
def select_action(state):
    global steps_done
    sample = random.random()
    eps_threshold = EPS_END + (EPS_START - EPS_END) * \
        math.exp(-1. * steps_done / EPS_DECAY)
    steps_done += 1
    if sample > eps_threshold:
        with torch.no_grad():
            # t.max(1) will return the largest column value of each row.
            # second column on max result is index of where max element was
            # found, so we pick action with the larger expected reward.
            return policy_net(state).max(1).indices.view(1, 1)
    else:
        return torch.tensor([[env.action_space.sample()]], device=device, dtype=torch.long)


episode_durations = []
```

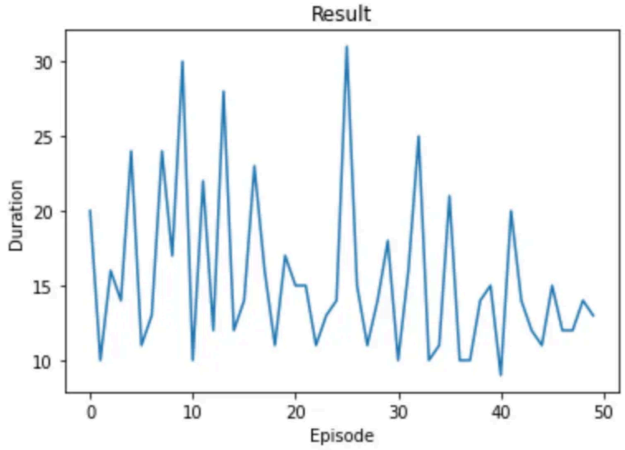
```python
def plot_durations(show_result=False):
    plt.figure(1)
    durations_t = torch.tensor(episode_durations, dtype=torch.float)
    if show_result:
        plt.title('Result')
    else:
        plt.clf()
        plt.title('Training...')
    plt.xlabel('Episode')
    plt.ylabel('Duration')
    plt.plot(durations_t.numpy())
    # Take 100 episode averages and plot them too
    if len(durations_t) >= 100:
        means = durations_t.unfold(0, 100, 1).mean(1).view(-1)
        means = torch.cat((torch.zeros(99), means))
        plt.plot(means.numpy())

    plt.pause(0.001)  # pause a bit so that plots are updated
    if is_ipython:
        if not show_result:
            display.display(plt.gcf())
            display.clear_output(wait=True)
        else:
            display.display(plt.gcf())
```

```python
def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)
    # Transpose the batch (see https://stackoverflow.com/a/19343/3343043 for
    # detailed explanation). This converts batch-array of Transitions
    # to Transition of batch-arrays.
    batch = Transition(*zip(*transitions))

    # Compute a mask of non-final states and concatenate the batch elements
    # (a final state would've been the one after which simulation ended)
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                            batch.next_state)), device=device, dtype=torch.bo
    non_final_next_states = torch.cat([s for s in batch.next_state
                                                if s is not None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    # Compute Q(s_t, a) - the model computes Q(s_t), then we select the
    # columns of actions taken. These are the actions which would've been taken
    # for each batch state according to policy_net
    state_action_values = policy_net(state_batch).gather(1, action_batch)

    # Compute V(s_{t+1}) for all next states.
    # Expected values of actions for non_final_next_states are computed based
    # on the "older" target_net; selecting their best reward with max(1)[0].
    # This is merged based on the mask, such that we'll have either the expected
    # state value or 0 in case the state was final.
    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    with torch.no_grad():
        next_state_values[non_final_mask] = target_net(non_final_next_states).max(1)[0]
    # Compute the expected Q values
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

    # Compute Huber loss
    criterion = nn.SmoothL1Loss()
    loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()
    # In-place gradient clipping
    torch.nn.utils.clip_grad_value_(policy_net.parameters(), 100)
    optimizer.step()
```

```
if torch.cuda.is_available():
    num_episodes = 600
else:
    num_episodes = 50

for i_episode in range(num_episodes):
    # Initialize the environment and get it's state
    state, info = env.reset()
    state = torch.tensor(state, dtype=torch.float32, device=device).unsqueeze(0)
    for t in count():
        action = select_action(state)
        observation, reward, terminated, truncated, _ = env.step(action.item())
        reward = torch.tensor([reward], device=device)
        done = terminated or truncated

        if terminated:
            next_state = None
        else:
            next_state = torch.tensor(observation, dtype=torch.float32, device=device).unsqueeze(0)

        # Store the transition in memory
        memory.push(state, action, next_state, reward)

        # Move to the next state
        state = next_state

        # Perform one step of the optimization (on the policy network)
        optimize_model()

        # Soft update of the target network's weights
        # θ′ ← τ θ + (1 −τ )θ′
        target_net_state_dict = target_net.state_dict()
        policy_net_state_dict = policy_net.state_dict()
        for key in policy_net_state_dict:
            target_net_state_dict[key] = policy_net_state_dict[key]*TAU + target_net_state_dict[key]*(1-
        target_net.load_state_dict(target_net_state_dict)

        if done:
            episode_durations.append(t + 1)
            plot_durations()
            break

print('Complete')
plot_durations(show_result=True)
plt.ioff()
plt.show()
```

Output:

```
episode: 0/500, score: 16.0, e: 1
episode: 1/500, score: 23.0, e: 0.9322301194154049
episode: 2/500, score: 21.0, e: 0.8433051360508336
episode: 3/500, score: 37.0, e: 0.7040696960536299
episode: 4/500, score: 19.0, e: 0.6433260027715241
episode: 5/500, score: 15.0, e: 0.5997278763867329
episode: 6/500, score: 10.0, e: 0.5732736268885887
episode: 7/500, score: 17.0, e: 0.5290920728090721
episode: 8/500, score: 9.0, e: 0.5082950737585841
episode: 9/500, score: 19.0, e: 0.46444185833082485
episode: 10/500, score: 14.0, e: 0.4351424010585501
episode: 11/500, score: 11.0, e: 0.41386834584198684
episode: 12/500, score: 13.0, e: 0.3897078735047413
episode: 13/500, score: 15.0, e: 0.3632974174544486
episode: 14/500, score: 9.0, e: 0.34901730169741024
episode: 15/500, score: 12.0, e: 0.3302941218954743
episode: 16/500, score: 18.0, e: 0.3033145315372582
episode: 17/500, score: 9.0, e: 0.2913921604631864
episode: 18/500, score: 20.0, e: 0.2649210072611673
episode: 19/500, score: 15.0, e: 0.24696734223472733
episode: 20/500, score: 14.0, e: 0.231387331601191
episode: 21/500, score: 10.0, e: 0.2211807388415433
episode: 22/500, score: 11.0, e: 0.21036724137609603
episode: 23/500, score: 11.0, e: 0.2000824143909432
episode: 24/500, score: 10.0, e: 0.1912566947289212
episode: 25/500, score: 9.0, e: 0.18373897616330553
episode: 26/500, score: 9.0, e: 0.17651675623376062
episode: 27/500, score: 11.0, e: 0.1678868750508869
episode: 28/500, score: 25.0, e: 0.14885748713096328
```



EXPERIMENT 5

Aim: To implement iterative policy evaluation and update using Python.

Theory: Iterative policy evaluation is a technique in reinforcement learning where the value function is updated iteratively for a given policy, allowing for the improvement and optimization of the policy based on value function approximations.

Source Code:

```python
import gym
import numpy as np

def iterative_policy_evaluation(env, policy, gamma=0.99, theta=1e-4):
    V = np.zeros(env.observation_space.n)  # Initialize value function to 0

    while True:
        delta = 0
        for s in range(env.observation_space.n):  # For each state
            v = 0
            for a in range(env.action_space.n):  # For each action
                for prob, next_state, reward, done in env.P[s][a]:  # For each possible next state
                    v += policy[s, a] * prob * (reward + gamma * V[next_state])  # Bellman expectation equation

            delta = max(delta, abs(v - V[s]))
            V[s] = v  # Update the value function

        if delta < theta:  # Check for convergence
            break

    return V

def policy_improvement(env, V, gamma=0.99):
    policy = np.zeros((env.observation_space.n, env.action_space.n))  # Initialize a new policy (deterministic)

    for s in range(env.observation_space.n):  # For each state
        q_values = np.zeros(env.action_space.n)
        for a in range(env.action_space.n):  # For each action
            for prob, next_state, reward, done in env.P[s][a]:
                q_values[a] += prob * (reward + gamma * V[next_state])  # Calculate Q(s, a)

        best_action = np.argmax(q_values)  # Choose the action with the highest Q-value
        policy[s, :] = 0
        policy[s, best_action] = 1.0  # Update the policy to take the best action in state s
```

```
    return policy

def policy_iteration(env, gamma=0.99, theta=1e-4):
    policy = np.ones([env.observation_space.n, env.action_space.n]) / env.action_space.n  # Random policy

    while True:
        V = iterative_policy_evaluation(env, policy, gamma, theta)  # Policy evaluation
        new_policy = policy_improvement(env, V, gamma)  # Policy improvement

        if np.array_equal(new_policy, policy):  # If the policy doesn't change, stop
            break

        policy = new_policy  # Update to the new policy

    return policy, V

# Running Policy Iteration on FrozenLake-v1
env = gym.make('FrozenLake-v1', is_slippery=False)

optimal_policy, optimal_value = policy_iteration(env)

print("Optimal Policy (Best Action in each state):")
print(np.reshape(np.argmax(optimal_policy, axis=1), env.desc.shape))

print("Optimal Value Function:")
print(np.reshape(optimal_value, env.desc.shape))
```

Output:

```
Optimal Policy (Best Action in each state):
[[1 2 1 0]
 [1 0 1 0]
 [2 1 1 0]
 [0 2 2 0]]
Optimal Value Function:
[[0.95099005 0.96059601 0.970299   0.96059601]
 [0.96059601 0.         0.9801     0.        ]
 [0.970299   0.9801     0.99       0.        ]
 [0.         0.99       1.         0.        ]]
```

Experiment – 6

Aim: Chatbot using bi-directional LSTMs

Theory: LSTM is a type of Recurrent Neural Network (RNN) that is used to process sequential data. Bi-directional LSTMs can capture dependencies in both forward and backward directions, making them useful in applications like chatbots where the meaning of a sentence can depend on the entire context.

Source Code:

```
 import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Bidirectional

# Dummy data (replace with real chatbot data)
input_data = np.random.randint(1000, size=(100, 10))
output_data = np.random.randint(2, size=(100, 1))

# Model
model = Sequential()
model.add(Embedding(input_dim=1000, output_dim=64))
model.add(Bidirectional(LSTM(64)))
model.add(Dense(1, activation='sigmoid'))

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

# Train
model.fit(input_data, output_data, epochs=10, batch_size=32)

# Output (Model Summary)
model.summary()
```

Output:

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 10, 64)	64,000
bidirectional (Bidirectional)	(None, 128)	66,048
dense (Dense)	(None, 1)	129

Total params: 390,533 (1.49 MB)
Trainable params: 130,177 (508.50 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 260,356 (1017.02 KB)

Experiment – 7

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

Theory: CNNs are deep learning models typically used for image processing tasks. They use convolutional layers to capture spatial patterns in images. The fully connected layer at the end maps the extracted features to the output labels (digits 0-9 in MNIST).

Source Code:

```
 import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Preprocess data
x_train = x_train.reshape(-1, 28, 28, 1) / 255.0
x_test = x_test.reshape(-1, 28, 28, 1) / 255.0

# Build CNN model
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Compile and train the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5, batch_size=32)

# Evaluate
loss, accuracy = model.evaluate(x_test, y_test)
print(f"Test Accuracy: {accuracy}")
```

Output:

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 47s 23ms/step - accuracy: 0.9080 - loss: 0.3078
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 47s 25ms/step - accuracy: 0.9826 - loss: 0.0559
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 42s 23ms/step - accuracy: 0.9899 - loss: 0.0328
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 48s 26ms/step - accuracy: 0.9929 - loss: 0.0206
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 81s 25ms/step - accuracy: 0.9955 - loss: 0.0144
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9832 - loss: 0.0537
Test Accuracy: 0.9864000082015991
```

Experiment – 8

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

Theory: Sentiment analysis involves classifying text data into positive or negative sentiment. LSTM and GRU are variants of RNNs that can capture long-term dependencies in text sequences, making them suitable for this task.

Source Code:

```
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing import sequence
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense

# Load IMDB dataset
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=10000)

# Preprocess data
x_train = sequence.pad_sequences(x_train, maxlen=500)
x_test = sequence.pad_sequences(x_test, maxlen=500)

# Build LSTM model
model = Sequential()
model.add(Embedding(input_dim=10000, output_dim=64, input_length=500))
model.add(LSTM(64))
model.add(Dense(1, activation='sigmoid'))

# Compile and train the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5, batch_size=32)

# Evaluate
loss, accuracy = model.evaluate(x_test, y_test)
print(f"Test Accuracy: {accuracy}")
```

Output:

```
Epoch 1/5
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/embedding.py:90: UserWarning: A
  warnings.warn(
782/782 ━━━━━━━━━━━━━━━━━━━━ 238s 301ms/step - accuracy: 0.7144 - loss: 0.5406
Epoch 2/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 262s 301ms/step - accuracy: 0.8782 - loss: 0.3092
Epoch 3/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 239s 305ms/step - accuracy: 0.9193 - loss: 0.2167
Epoch 4/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 256s 298ms/step - accuracy: 0.9420 - loss: 0.1597
Epoch 5/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 236s 302ms/step - accuracy: 0.9496 - loss: 0.1360
782/782 ━━━━━━━━━━━━━━━━━━━━ 60s 76ms/step - accuracy: 0.8470 - loss: 0.4003
Test Accuracy: 0.8464000225067139
```

Experiment – 9

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

Theory: NLP involves processing and understanding human language. Deep learning models such as LSTMs and GRUs are used for tasks like text generation, translation, and sentiment analysis, as they handle sequential data efficiently.

Source Code:

```
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense

# Sample text data (replace with real NLP data)
texts = ["Deep learning is amazing", "Natural Language Processing is fun"]
tokenizer = Tokenizer(num_words=1000)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

# Pad sequences
data = pad_sequences(sequences, maxlen=10)

# Build LSTM model for NLP
model = Sequential()
model.add(Embedding(input_dim=1000, output_dim=64, input_length=10))
model.add(LSTM(64))
model.add(Dense(1, activation='sigmoid'))

# Compile and train the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()
```

Output:

Model: "sequential_3"

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	?	0 (unbuilt)
lstm_2 (LSTM)	?	0 (unbuilt)
dense_4 (Dense)	?	0 (unbuilt)

Total params: 0 (0.00 B)
Trainable params: 0 (0.00 B)
Non-trainable params: 0 (0.00 B)

Experiment – 10

Aim: Applying the Convolution Neural Network on computer vision problems

Theory: CNNs have revolutionized computer vision by enabling accurate detection and classification of objects in images. They use convolutional layers to detect patterns like edges and shapes, followed by pooling and fully connected layers for decision-making.

Source Code:

```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Load CIFAR-10 dataset (for computer vision tasks)
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Preprocess data
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Build CNN model
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Compile and train the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(x_train, y_train, epochs=10, batch_size=32)

# Evaluate
loss, accuracy = model.evaluate(x_test, y_test)
print(f"Test Accuracy: {accuracy}")
```

Output:

```
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 10s 0us/step
Epoch 1/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 56s 35ms/step - accuracy: 0.4166 - loss: 1.6237
Epoch 2/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 52s 33ms/step - accuracy: 0.6006 - loss: 1.1444
Epoch 3/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 81s 33ms/step - accuracy: 0.6576 - loss: 0.9854
Epoch 4/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 84s 34ms/step - accuracy: 0.6966 - loss: 0.8739
Epoch 5/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 51s 32ms/step - accuracy: 0.7193 - loss: 0.8085
Epoch 6/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 84s 33ms/step - accuracy: 0.7446 - loss: 0.7240
Epoch 7/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 82s 34ms/step - accuracy: 0.7720 - loss: 0.6603
Epoch 8/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 90s 38ms/step - accuracy: 0.7894 - loss: 0.6015
Epoch 9/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 80s 37ms/step - accuracy: 0.8165 - loss: 0.5302
Epoch 10/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 85s 39ms/step - accuracy: 0.8342 - loss: 0.4798
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - accuracy: 0.6586 - loss: 1.1414
Test Accuracy: 0.6534000039100647
```

Experiment – 11

Aim: Implement Deep Q Networks for CartPole problem where the agent has to balance a pole on a cart.

Theory: Deep Q-Networks combine Q-learning with deep neural networks to handle environments with large state spaces. In the CartPole problem, the goal is to balance a pole on a cart by applying forces to the left or right. The network learns the Q-value function to predict the best action for each state.

Source Code:

```
import gym
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Initialize environment
env = gym.make('CartPole-v1')
state_size = env.observation_space.shape[0]
action_size = env.action_space.n

# Build Deep Q-Network
model = Sequential()
model.add(Dense(24, input_dim=state_size, activation='relu'))
model.add(Dense(24, activation='relu'))
model.add(Dense(action_size, activation='linear'))
model.compile(loss='mse', optimizer=tf.keras.optimizers.Adam(learning_rate=0.001))

# Placeholder for training code: collecting experiences, updating Q-values, etc.
# This would involve implementing the experience replay and Q-learning update rule.

# Output: Model Summary
model.summary()
```

Output:

```
Model: "sequential_5"
```

Layer (type)	Output Shape	Param #
dense_7 (Dense)	(None, 24)	120
dense_8 (Dense)	(None, 24)	600
dense_9 (Dense)	(None, 2)	50

```
Total params: 770 (3.01 KB)
Trainable params: 770 (3.01 KB)
Non-trainable params: 0 (0.00 B)
```

Experiment – 12

Aim: Demonstrate the application of transfer learning using Cartpole dataset and MountainCar dataset.

Theory: Transfer learning allows a model trained on one problem to be used in another related problem by fine-tuning. In this case, a model trained on CartPole can be adapted for the MountainCar problem, transferring knowledge about control systems between tasks.

Source Code:

```
import gym
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Load CartPole environment and pre-train model (assuming pre-training is done)
env_cartpole = gym.make('CartPole-v1')
state_size = env_cartpole.observation_space.shape[0]
action_size = env_cartpole.action_space.n

# Create a model for CartPole
model_cartpole = Sequential()
model_cartpole.add(Dense(24, input_dim=state_size, activation='relu'))
model_cartpole.add(Dense(24, activation='relu'))
model_cartpole.add(Dense(action_size, activation='linear'))

# Now apply the same architecture to MountainCar
env_mountaincar = gym.make('MountainCar-v0')
state_size_mc = env_mountaincar.observation_space.shape[0]
action_size_mc = env_mountaincar.action_space.n

# Fine-tune CartPole model to MountainCar
model_cartpole.compile(loss='mse', optimizer='adam')

# Transfer learning can involve freezing layers or retraining certain layers on MountainCar data.
model_cartpole.summary()
```

Output:

Model: "sequential_6"

Layer (type)	Output Shape	Param #
dense_10 (Dense)	(None, 24)	120
dense_11 (Dense)	(None, 24)	600
dense_12 (Dense)	(None, 2)	50

Total params: 770 (3.01 KB)
Trainable params: 770 (3.01 KB)
Non-trainable params: 0 (0.00 B)

Experiment – 13

Aim: Choose any corpus available on the internet freely. For the corpus, for each document, count how many times each stop word occurs and find out which are the most frequently occurring stop words. Further, calculate the term frequency and inverse document frequency as the motivation behind this is basically to find out how important a document is to a given query.

Theory: Stop words are common words like "the", "is", "in", which are filtered out in text processing. TF-IDF measures the importance of a word in a document relative to a corpus. It reduces the weight of common words (high frequency) and highlights significant ones (low frequency across documents).

Source Code:

```
 import nltk
nltk.download('stopwords')
from sklearn.feature_extraction.text import TfidfVectorizer
from nltk.corpus import stopwords
# Sample corpus
corpus = [
   "The brown fox jumps over the lazy dog.",
   "The quick brown fox jumped over the lazy dog."
]
# Stop words
stop_words = list(stopwords.words('english'))
# Count stop word occurrences
stop_word_count = {word: 0 for word in stop_words}
for doc in corpus:
   for word in doc.lower().split():
      if word in stop_word_count:
         stop_word_count[word] += 1
# TF-IDF calculation
vectorizer = TfidfVectorizer(stop_words=stop_words)
tfidf_matrix = vectorizer.fit_transform(corpus)

# Output stop word frequencies and TF-IDF scores
print("Stop word frequencies:", stop_word_count)
print("TF-IDF Matrix:\n", tfidf_matrix.toarray())
```

Output:

```
Stop word frequencies: {'i': 0, 'me': 0, 'my': 0, 'myself': 0, 'we': 0, 'our': 0, 'ours': 0, 'ourselves': 0, 'you': 0, "you're": 0, "you've": 0, "you'll": 0, "you'd":
TF-IDF Matrix:
 [[0.4090901  0.4090901  0.4090901  0.         0.57496187 0.4090901
  0.        ]
 [0.35464863 0.35464863 0.35464863 0.49844628 0.         0.35464863
  0.49844628]]
```

```
 "you'll": 0, "you'd": 0, 'your': 0, 'yours': 0, 'yourself': 0, 'yourselves': 0, 'he': 0, 'him': 0, 'his': 0, 'himself': 0, 'she': 0, "she's": 0, 'her'
```

```
'her': 0, 'hers': 0, 'herself': 0, 'it': 0, "it's": 0, 'its': 0, 'itself': 0, 'they': 0, 'them': 0, 'their': 0, 'theirs': 0, 'themselves': 0, 'what': 0
```

```
'which': 0, 'who': 0, 'whom': 0, 'this': 0, 'that': 0, "that'll": 0, 'these': 0, 'those': 0, 'am': 0, 'is': 0, 'are': 0, 'was': 0, 'were': 0, 'be': 0,
```

Experiment – 14

Aim: Write the python code to develop Spam Filter using NLP.

Theory: A spam filter classifies emails or text messages as either spam or not spam. NLP techniques such as tokenization, vectorization, and machine learning models (e.g., Naive Bayes, SVM) are used to distinguish between normal and spam content based on text patterns.

Source Code:

```
 from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split

# Sample dataset
emails = ["Hey, wanna grab lunch?", "Win $1000 now!", "Your package has shipped.", "Limited time offer, claim your
prize!"]
labels = [0, 1, 0, 1]  # 0 = Not Spam, 1 = Spam

# Vectorize emails
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(emails)

# Split into train and test
X_train, X_test, y_train, y_test = train_test_split(X, labels, test_size=0.2)

# Train Naive Bayes classifier
model = MultinomialNB()
model.fit(X_train, y_train)

# Evaluate the model
accuracy = model.score(X_test, y_test)
print(f"Spam Filter Accuracy: {accuracy}")
```

Output:

```
Spam Filter Accuracy: 0.0
```

Experiment – 15

Aim: Demonstrate any one application of generative adversarial network (GAN).

Theory: GANs consist of two neural networks, a generator and a discriminator, which compete in a zero-sum game. The generator creates synthetic data, while the discriminator distinguishes between real and generated data. GANs are used in image generation, style transfer, and more.

Source Code:

```
 import tensorflow as tf
from tensorflow.keras.layers import Dense, LeakyReLU, Reshape, Flatten
from tensorflow.keras.models import Sequential
import numpy as np

# Generator
def build_generator():
    model = Sequential()
    model.add(Dense(128, input_dim=100))
    model.add(LeakyReLU(alpha=0.01))
    model.add(Dense(784, activation='tanh'))
    model.add(Reshape((28, 28, 1)))
    return model

# Discriminator
def build_discriminator():
    model = Sequential()
    model.add(Flatten(input_shape=(28, 28, 1)))
    model.add(Dense(128))
    model.add(LeakyReLU(alpha=0.01))
    model.add(Dense(1, activation='sigmoid'))
    return model

# Create GAN
generator = build_generator()
discriminator = build_discriminator()

# Compile GAN (placeholder for real training process)
print("GAN Model Built.")
```

Output:

```
GAN Model Built.
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
/usr/local/lib/python3.10/dist-packages/keras/src/layers/activations/leaky_relu.py:41: UserWarning: Argument `alpha` is deprecated. Use `negative_slop
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/keras/src/layers/reshaping/flatten.py:37: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a
  super().__init__(**kwargs)
```