# Import necessary libraries

import os

import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim

import gym

from torchvision import transforms, datasets

from torch.utils.data import DataLoader

from scipy.spatial.distance import euclidean

from google.colab import files

import zipfile

# ==========================

# 📌 Step 1: Upload & Extract Dataset

# ==========================

# Upload dataset ZIP file

uploaded = files.upload()  # Prompts user to upload a file

# Extract ZIP File

for filename in uploaded.keys():

    if filename.endswith(".zip"):

        zip\_ref = zipfile.ZipFile(filename, 'r')

        zip\_ref.extractall("/content/dataset")  # Extract to /content/dataset

        zip\_ref.close()

        os.remove(filename)  # Remove ZIP file after extraction

data\_dir = "/content/dataset"  # Dataset path

# ==========================

# ✅ Step 2: Validate Dataset

# ==========================

valid\_extensions = {".jpg", ".jpeg", ".png"}

# Check if dataset exists & contains valid images

if not os.path.exists(data\_dir) or len(os.listdir(data\_dir)) == 0:

    raise FileNotFoundError(f"Dataset not found in {data\_dir}. Please upload a valid dataset.")

for root, \_, files in os.walk(data\_dir):

    for file in files:

        if not file.lower().endswith(tuple(valid\_extensions)):

            raise ValueError(f"Invalid file found: {file}. Expected image files (.jpg, .jpeg, .png).")

# ==========================

# 📌 Step 3: Load Dataset for Training

# ==========================

# Image transformations

transform = transforms.Compose([

    transforms.Grayscale(),

    transforms.Resize((64, 64)),

    transforms.ToTensor()

])

# Load dataset

dataset = datasets.ImageFolder(root=data\_dir, transform=transform)

dataloader = DataLoader(dataset, batch\_size=32, shuffle=True)

# ==========================

# ✅ Step 4: Define Deep Q-Network (DQN)

# ==========================

class DQN(nn.Module):

    def \_\_init\_\_(self, input\_dim, output\_dim):

        super(DQN, self).\_\_init\_\_()

        self.fc1 = nn.Linear(input\_dim, 128)

        self.fc2 = nn.Linear(128, 128)

        self.fc3 = nn.Linear(128, output\_dim)

    def forward(self, x):

        x = torch.flatten(x, start\_dim=1)  # Flatten image input

        x = torch.relu(self.fc1(x))

        x = torch.relu(self.fc2(x))

        x = self.fc3(x)

        return x

# ==========================

# 🔥 Step 5: Firefly Algorithm for Optimization

# ==========================

def firefly\_algorithm(n\_fireflies, max\_iter, obj\_func, dim):

    alpha = 0.5  # Randomness strength

    beta\_0 = 1.0  # Attraction factor

    gamma = 1.0  # Absorption coefficient

    fireflies = np.random.rand(n\_fireflies, dim)

    best\_solution = fireflies[0]

    best\_score = float('inf')

    for \_ in range(max\_iter):

        for i in range(n\_fireflies):

            for j in range(n\_fireflies):

                if obj\_func(fireflies[j]) < obj\_func(fireflies[i]):

                    r = euclidean(fireflies[i], fireflies[j])

                    beta = beta\_0 \* np.exp(-gamma \* r \*\* 2)

                    fireflies[i] += beta \* (fireflies[j] - fireflies[i]) + alpha \* (np.random.rand(dim) - 0.5)

                    fireflies[i] = np.clip(fireflies[i], 0, 1)

                    if obj\_func(fireflies[i]) < best\_score:

                        best\_score = obj\_func(fireflies[i])

                        best\_solution = fireflies[i]

    return best\_solution

# ==========================

# 🎯 Step 6: Define Custom Facial Recognition RL Environment

# ==========================

class FacialRecEnv(gym.Env):

    def \_\_init\_\_(self):

        super(FacialRecEnv, self).\_\_init\_\_()

        self.observation\_space = gym.spaces.Box(low=0, high=1, shape=(64, 64), dtype=np.float32)

        self.action\_space = gym.spaces.Discrete(2)  # Accept (1) or Reject (0)

    def reset(self):

        return np.random.rand(64, 64)

    def step(self, action):

        reward = 1 if action == np.random.choice([0, 1]) else -1  # Random reward logic

        return np.random.rand(64, 64), reward, False, {}

# ==========================

# 🚀 Step 7: Train DQN with Firefly Algorithm

# ==========================

def train\_dqn():

    env = FacialRecEnv()

    dqn = DQN(64 \* 64, env.action\_space.n)

    optimizer = optim.Adam(dqn.parameters(), lr=0.001)

    loss\_fn = nn.MSELoss()

    gamma = 0.9  # Discount factor

    memory = []  # Replay memory buffer

    for episode in range(50):  # Train for 50 episodes

        state = env.reset()

        state = torch.tensor(state.flatten(), dtype=torch.float32).unsqueeze(0)

        done = False

        while not done:

            q\_values = dqn(state)

            action = torch.argmax(q\_values).item()

            next\_state, reward, done, \_ = env.step(action)

            next\_state = torch.tensor(next\_state.flatten(), dtype=torch.float32).unsqueeze(0)

            memory.append((state, action, reward, next\_state))

            # Train the model if memory has enough samples

            if len(memory) > 10:

                batch = np.random.choice(memory, 10, replace=False)

                batch\_states, batch\_actions, batch\_rewards, batch\_next\_states = zip(\*batch)

                batch\_states = torch.cat(batch\_states)

                batch\_next\_states = torch.cat(batch\_next\_states)

                batch\_rewards = torch.tensor(batch\_rewards, dtype=torch.float32)

                target\_qs = batch\_rewards + gamma \* torch.max(dqn(batch\_next\_states), dim=1)[0]

                predicted\_qs = dqn(batch\_states).gather(1, torch.tensor(batch\_actions).unsqueeze(1)).squeeze()

                loss = loss\_fn(predicted\_qs, target\_qs)

                optimizer.zero\_grad()

                loss.backward()

                optimizer.step()

        # Optimize DQN weights using Firefly Algorithm

        def obj\_func(weights):

            with torch.no\_grad():

                dqn.fc1.weight.data = torch.tensor(weights[:128].reshape(128, -1))

                dqn.fc2.weight.data = torch.tensor(weights[128:256].reshape(128, -1))

                dqn.fc3.weight.data = torch.tensor(weights[256:].reshape(2, -1))

                return -torch.mean(dqn(state)).item()

        best\_weights = firefly\_algorithm(10, 20, obj\_func, 258)

        print(f"Episode {episode}: Firefly Algorithm optimized the Q-values.")

    torch.save(dqn.state\_dict(), "dqn\_model.pth")

    print("Model saved as dqn\_model.pth")

# ==========================

# ✅ Step 8: Run Training

# ==========================

train\_dqn()