import pygame

import math

import random

# Constants for screen size

SIZE\_OF\_THE\_SCREEN = 424, 430

# Dimensions of the bricks

HEIGHT\_OF\_BRICK = 13

WIDTH\_OF\_BRICK = 32

# Dimensions of the paddle

HEIGHT\_OF\_PADDLE = 8

PADDLE\_WIDTH = 50

# Y coordinate for Paddle

PADDLE\_Y = SIZE\_OF\_THE\_SCREEN[1] - HEIGHT\_OF\_PADDLE - 10

# Dimensions of the ball

BALL\_DIAMETER = 12

BALL\_RADIUS = BALL\_DIAMETER // 2

# X coordinate for Paddle and Ball

MAX\_PADDLE\_X = SIZE\_OF\_THE\_SCREEN[0] - PADDLE\_WIDTH

MAX\_BALL\_X = SIZE\_OF\_THE\_SCREEN[0] - BALL\_DIAMETER

MAX\_BALL\_Y = SIZE\_OF\_THE\_SCREEN[1] - BALL\_DIAMETER

# Color constants

BLACK = (0, 0, 0)

WHITE = (255, 255, 255)

BLUE = (0, 0, 255)

COLOR\_OF\_BRICK = (153, 76, 0)

PADDLE\_COLOR = (204, 0, 0)

FPS = 60

FPSCLOCK = pygame.time.Clock()

pygame.init()

screen = pygame.display.set\_mode(SIZE\_OF\_THE\_SCREEN)

pygame.display.set\_caption("BREAKOUT")

clock = pygame.time.Clock()

class Breakout:

def \_\_init\_\_(self):

self.capture = 0

self.ball\_vel = [12, -12]

self.paddle = pygame.Rect(215, PADDLE\_Y, PADDLE\_WIDTH, HEIGHT\_OF\_PADDLE)

self.ball = pygame.Rect(225, PADDLE\_Y - BALL\_DIAMETER, BALL\_DIAMETER, BALL\_DIAMETER)

self.create\_bricks()

def create\_bricks(self):

y\_ofs = 20

self.bricks = []

for i in range(11):

x\_ofs = 15

for j in range(12):

self.bricks.append(pygame.Rect(x\_ofs, y\_ofs, WIDTH\_OF\_BRICK, HEIGHT\_OF\_BRICK))

x\_ofs += WIDTH\_OF\_BRICK + 1

y\_ofs += HEIGHT\_OF\_BRICK + 1

def draw\_bricks(self):

for brick in self.bricks:

pygame.draw.rect(screen, COLOR\_OF\_BRICK, brick)

def draw\_paddle(self):

pygame.draw.rect(screen, PADDLE\_COLOR, self.paddle)

def draw\_ball(self):

pygame.draw.circle(screen, WHITE, (self.ball.left + BALL\_RADIUS, self.ball.top + BALL\_RADIUS), BALL\_RADIUS)

def check\_input(self, input\_action):

# 0-LEFT, 1-Right

if input\_action[0] == 1:

self.paddle.left -= 12

if self.paddle.left < 0:

self.paddle.left = 0

if input\_action[1] == 1:

self.paddle.left += 12

if self.paddle.left > MAX\_PADDLE\_X:

self.paddle.left = MAX\_PADDLE\_X

def move\_ball(self):

self.ball.left += self.ball\_vel[0]

self.ball.top += self.ball\_vel[1]

if self.ball.left <= 0:

self.ball.left = 0

self.ball\_vel[0] = -self.ball\_vel[0]

elif self.ball.left >= MAX\_BALL\_X:

self.ball.left = MAX\_BALL\_X

self.ball\_vel[0] = -self.ball\_vel[0]

if self.ball.top < 0:

self.ball.top = 0

self.ball\_vel[1] = -self.ball\_vel[1]

elif self.ball.top >= MAX\_BALL\_Y:

self.ball.top = MAX\_BALL\_Y

self.ball\_vel[1] = -self.ball\_vel[1]

def take\_action(self, input\_action):

pygame.event.pump()

reward = 0.1

terminal = False

for event in pygame.event.get():

if event.type == pygame.QUIT:

pygame.quit()

quit()

screen.fill(BLACK)

self.check\_input(input\_action)

self.move\_ball()

for brick in self.bricks:

if self.ball.colliderect(brick):

reward = 2

self.ball\_vel[1] = -self.ball\_vel[1]

self.bricks.remove(brick)

break

if len(self.bricks) == 0:

terminal = True

self.\_\_init\_\_()

if self.ball.colliderect(self.paddle):

self.ball.top = PADDLE\_Y - BALL\_DIAMETER

self.ball\_vel[1] = -self.ball\_vel[1]

elif self.ball.top > self.paddle.top:

terminal = True

self.\_\_init\_\_()

reward = -2

self.draw\_bricks()

self.draw\_ball()

self.draw\_paddle()

image\_data = pygame.surfarray.array3d(pygame.display.get\_surface())

pygame.display.update()

FPSCLOCK.tick(FPS)

return image\_data, reward, terminal

import cv2

import numpy as np

from collections import deque

import torch

import torch.nn as nn

import torch.optim as optim

import torch.nn.functional as F

import os

import sys

import time

# Check if GPU is available

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

class NeuralNetwork(nn.Module):

def \_\_init\_\_(self):

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

self.number\_of\_actions = 2

self.gamma = 0.99

self.final\_epsilon = 0.05

self.initial\_epsilon = 0.1

self.number\_of\_iterations = 2000000

self.replay\_memory\_size = 750000

self.minibatch\_size = 32

self.explore = 3000000

self.conv1 = nn.Conv2d(4, 32, kernel\_size=8, stride=4)

self.conv2 = nn.Conv2d(32, 64, kernel\_size=4, stride=2)

self.conv3 = nn.Conv2d(64, 64, kernel\_size=3, stride=1)

self.fc4 = nn.Linear(7 \* 7 \* 64, 512)

self.fc5 = nn.Linear(512, self.number\_of\_actions)

def forward(self, x):

x = F.relu(self.conv1(x))

x = F.relu(self.conv2(x))

x = F.relu(self.conv3(x))

x = F.relu(self.fc4(x.view(x.size(0), -1)))

return self.fc5(x)

def preprocessing(image):

image\_data = cv2.cvtColor(cv2.resize(image, (84, 84)), cv2.COLOR\_BGR2GRAY)

image\_data[image\_data > 0] = 255

image\_data = np.reshape(image\_data, (84, 84, 1))

image\_tensor = image\_data.transpose(2, 0, 1)

image\_tensor = image\_tensor.astype(np.float32)

image\_tensor = torch.from\_numpy(image\_tensor).to(device)

return image\_tensor

def init\_weights(m):

if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):

torch.nn.init.uniform\_(m.weight, -0.01, 0.01)

m.bias.data.fill\_(0.01)

def train(model, start):

optimizer = optim.Adam(model.parameters(), lr=0.0002)

criterion = nn.MSELoss()

game\_state = Breakout()

D = deque()

action = torch.zeros([model.number\_of\_actions], dtype=torch.float32).to(device)

action[0] = 1

image\_data, reward, terminal = game\_state.take\_action(action.cpu().numpy())

image\_data = preprocessing(image\_data)

state = torch.cat((image\_data, image\_data, image\_data, image\_data)).unsqueeze(0).to(device)

epsilon = model.initial\_epsilon

iteration = 0

while iteration < model.number\_of\_iterations:

output = model(state)[0]

action = torch.zeros([model.number\_of\_actions], dtype=torch.float32).to(device)

random\_action = random.random() <= epsilon

if random\_action:

action\_index = torch.randint(model.number\_of\_actions, torch.Size([]), dtype=torch.int).to(device)

else:

action\_index = torch.argmax(output)

action[action\_index] = 1

if epsilon > model.final\_epsilon:

epsilon -= (model.initial\_epsilon - model.final\_epsilon) / model.explore

image\_data\_1, reward, terminal = game\_state.take\_action(action.cpu().numpy())

image\_data\_1 = preprocessing(image\_data\_1)

state\_1 = torch.cat((state.squeeze(0)[1:, :, :], image\_data\_1)).unsqueeze(0).to(device)

action = action.unsqueeze(0)

reward = torch.tensor([reward], dtype=torch.float32).unsqueeze(0).to(device)

D.append((state, action, reward, state\_1, terminal))

if len(D) > model.replay\_memory\_size:

D.popleft()

minibatch = random.sample(D, min(len(D), model.minibatch\_size))

state\_batch = torch.cat(tuple(d[0] for d in minibatch))

action\_batch = torch.cat(tuple(d[1] for d in minibatch))

reward\_batch = torch.cat(tuple(d[2] for d in minibatch))

state\_1\_batch = torch.cat(tuple(d[3] for d in minibatch))

output\_1\_batch = model(state\_1\_batch)

y\_batch = torch.cat(tuple(reward\_batch[i] if minibatch[i][4]

else reward\_batch[i] + model.gamma \* torch.max(output\_1\_batch[i])

for i in range(len(minibatch))))

q\_value = torch.sum(model(state\_batch) \* action\_batch, dim=1)

optimizer.zero\_grad()

y\_batch = y\_batch.detach()

loss = criterion(q\_value, y\_batch)

loss.backward()

optimizer.step()

state = state\_1

iteration += 1

if iteration % 10000 == 0:

torch.save(model, f"trained\_model/current\_model\_{iteration}.pth")

print(f"total iteration: {iteration} Elapsed time: {(time.time() - start) / 60:.2f} epsilon: {epsilon:.5f}"

f" action: {action\_index.cpu().numpy()} Reward: {reward.cpu().numpy()[0]}")

def test(model):

game\_state = Breakout()

action = torch.zeros([model.number\_of\_actions], dtype=torch.float32).to(device)

action[0] = 1

image\_data, reward, terminal = game\_state.take\_action(action.cpu().numpy())

image\_data = preprocessing(image\_data)

state = torch.cat((image\_data, image\_data, image\_data, image\_data)).unsqueeze(0).to(device)

while True:

output = model(state)[0]

action = torch.zeros([model.number\_of\_actions], dtype=torch.float32).to(device)

action\_index = torch.argmax(output)

action[action\_index] = 1

image\_data\_1, reward, terminal = game\_state.take\_action(action.cpu().numpy())

image\_data\_1 = preprocessing(image\_data\_1)

state\_1 = torch.cat((state.squeeze(0)[1:, :, :], image\_data\_1)).unsqueeze(0).to(device)

state = state\_1

def main(mode):

if mode == 'test':

model = torch.load('trained\_model/current\_model\_430000.pth', map\_location=device).eval()

test(model)

elif mode == 'train':

if not os.path.exists('trained\_model/'):

os.mkdir('trained\_model/')

model = NeuralNetwork().to(device)

model.apply(init\_weights)

start = time.time()

train(model, start)

elif mode == 'continue':

model = torch.load('trained\_model/current\_model\_430000.pth', map\_location=device).eval()

start = time.time()

train(model, start)

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

main(sys.argv[1])