```
#!/bin/python3.6.7
# Ostap Voynarovskiy and Luka Lipovac
# CGML Midterm
# October 21 2018
# Professor Curro

# Paper we were implementing
# We looked at Rainbow and wanted to implement some of the
# Algorithims in it
# We ended up choosing DQN, DDQN, D3QN
# https://arxiv.org/pdf/1710.02298.pdf
# The DQN specifics were taken from the paper that introduced it
# https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf
# The hacky paper we implemented to get the dqn to work for continuous output sp
aces
# http://users.cecs.anu.edu.au/~rsl/rsl_papers/99ai.kambara.pdf?fbclid=IwAR1hAir
s1uX9Ya21PAR6kVRcT7_ivK5YtTm4w5KPasH2iRPtffMdSpF8Wh0

# Documentation:
# We chose to implement parts of the Rainbow paper for this midterm
# The benchmark for many reinforcement learning aproaches is the
# open ai gym. Since Ben Ma was going to be doing the inverted penulum,
# we decided to pick a different problem. We chose QWOP initially
# thinking that it would provide us with a limited number of state
# variables and actions. We hoped this would reduce training time as
# compared to something like PAC-MAN. What we failed to realize however
# is that The QWOP we know and the bipedal walker api provided by
# open ai gym were not the same. While the Real game provides 4 discrete
# inputs which are boolean, the open ai gym provides 4 floats from -1 to 1
# corresponding to the torque delivered by each of the 4 joints.
# Initially we didn't think much of this but quickly we realized that
# the premise of deep Q learning requires a discrete action space.
# Being that we had spent a lot of time setting up the environment,
# we decided to try a hack that we read about in the third paper
# mentioned above, where we could discretize the outputs. We had high
# hopes for this since it would essentially be doing what qwop does,
# however after 15 hours training on a gtx 980 with an i7 4790k cpu,
# it managed to sometimes split its legs and not fall over. Reailizing
# we weren't making progress, we switched environments to Breakout-ram-v0
# where we implemented the DQN, dueling DQN and double DQN with Huber loss.
# Our final model was a combination of the models we had implemented. To see if
# it would converge, we let it also train on Cartpole, we achieved
# convergence in less than 10 minutes, so we are hoping to see good
# results from breakout. According to the internet, similar models have
# taken aproximately 35 hours to train, so we will just have to wait.

# these guys helped us through DQN
# https://keon.io/deep-q-learning/
# https://becominghuman.ai/lets-build-an-atari-ai-part-1-dqn-df57e8ff3b26

import gym
import time
import keras
import random
import pylab
import numpy
import matplotlib.pyplot as plt
from gym import wrappers
import numpy as np
import pandas as pd
from collections import deque
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
from keras import backend as K
from argparse import ArgumentParser

EPISODES = 4000000
```

```
#https://becominghuman.ai/beat-atari-with-deep-reinforcement-learning-part-2-dqn
-improvements-d3563f665a2c
def huber_loss(a, b, in_keras=True):
    error = a - b
    quadratic_term = error*error / 2
    linear_term = abs(error) - 1/2
    use_linear_term = (abs(error) > 1.0)
    if in_keras:
        use_linear_term = K.cast(use_linear_term, 'float32')
    return use_linear_term * linear_term + (1-use_linear_term) * quadratic_term

class D3QN:
        def __init__(self, stateSize,actionSize):
                self.stateSize = stateSize
                self.actionSize = actionSize

                self.memory = deque(maxlen=10000000)
                self.gamma = .99
                self.tau = 0.125
                self.epsilon = 1.0
                self.epsilonMin = 0.01 #1 percent random actions
                annealedFrames = 250000
                self.epsilonDecay = (self.epsilon-self.epsilonMin)/annealedFrame
s
                self.learningRate = 0.0000625
                self.adamEpsilon = 0.00015

                self.trainingStart = 200000
                self.batchSize = 64

                self.model = self.buildModel()
                self.targetModel = self.buildModel()

                self.updateTargetModel()

        def buildModel(self):
                model = Sequential()
                model.add(Dense(128, input_dim=self.stateSize, activation="relu")
)
                model.add(Dense(32, activation="relu"))
                model.add(Dense(self.actionSize, activation="linear")) #no activat
ion this is a regression
                model.compile(loss=huber_loss,optimizer=keras.optimizers.Adam(lr
=self.learningRate, epsilon=self.adamEpsilon))

                return model

        def remember(self, state, action, reward, nextState, done):
                self.memory.append((state, action, reward, nextState, done))

        def act(self, state):
                if np.random.uniform() < self.epsilon:
                        return random.randrange(self.actionSize)
                else:
                        return self.model.predict(state).argmax()

        def replay(self):
                if len(self.memory)<self.trainingStart:
                        return

                batches = random.sample(self.memory, self.batchSize)

                for batch in batches:
                        state, action, reward, nextState, done = batch

                        target = self.targetModel.predict(state)
                        if done:

                                target[0][action] = reward
```

```
			else:
				qFuture = np.argmax(self.model.predict(nextState
)[0])
				qTarget = self.targetModel.predict(nextState)[0]
				target[0][action] = reward + self.gamma*qTarget[
qFuture] #add discounted award

			self.model.fit(state,target,epochs=1,verbose=0)

		if self.epsilon > self.epsilonMin:
			self.epsilon -= self.epsilonDecay

	def updateTargetModel(self):
		weights = self.model.get_weights()
		targetWeights = self.targetModel.get_weights()
		for i in range(len(targetWeights)):
			targetWeights[i] = weights[i] * self.tau + targetWeights
[i] * (1 - self.tau)

		self.targetModel.set_weights(targetWeights)


	def daftPunk(self, rw): #read = true #just for saving and loading model
weights
		#Write it, cut it, paste it, save it,
		#Load it, check it, quick, rewrite it
		if rw:
			#self.model.load_weights(filename)
			print("load success")
		else:
			self.model.save_weights("./models/"+args.fileName+"_"+str(t
ime.time())+"-Breakout.csv")


def main():
	env = gym.make('Breakout-ram-v0')
	stateSize = env.observation_space.shape[0]
	actionSize = env.action_space.n

	agent = D3QN(stateSize,actionSize)

	episodes, scores, epsilon = [], [], []

	for ep in range(EPISODES):
		done = False
		state = env.reset()
		state = np.reshape(state,[1,stateSize])

		score = 0
		lives = 5

		while not done:
			if ep%args.renderAmount == 0:
				env.render()

			action = agent.act(state)

			nextState, reward, done, info = env.step(action)
			nextState = np.reshape(nextState, [1,stateSize])
			#reward = reward if not done else -10 #penialize for dyi
ng

			agent.remember(state,action,reward,nextState,done)
			agent.replay()

			state = nextState
			score += reward

		if done:
```

```
			scores.append(score)
			episodes.append(ep)
			epsilon.append(agent.epsilon)

			if ep%25 == 0:
				pylab.plot(episodes, scores, 'b')
				pylab.xlabel('Episodes')
				pylab.ylabel('Score')
				pylab.title('Breakout: Episodes vs Score')
				pylab.savefig("./"+args.fileName+"-Breakout.pdf")

				dataBreakout = pd.DataFrame({'Episode':episodes,
'Score':scores, 'Epsilon':epsilon})
				dataBreakout.to_csv("./"+args.fileName+"-Breakout.cs
v")

				agent.daftPunk(0)

			print("Episode: {}/{}, score: {}, epsilon:{:.4}".format(ep,EPISODES,s
core,agent.epsilon))
		agent.updateTargetModel()


parser = ArgumentParser()
parser.add_argument("-o", "--output", dest="fileName", default="test")
parser.add_argument("-r", "--render", dest="renderAmount", type=int, default=1)

args = parser.parse_args()

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