10 Mayıs 2019

İbrahim Doğan – hANIFI ENES GUL – ATAKAN KAYA

CMPE 373 AGENT BASED MODELING

DEEP REINFORCEMENT LEARNING wıth openaı gym

Project Title:

Deep Reinforcement Learning with OpenAI Gym (CartPole-v0)

Group Name:

SıfırSıfır7

Students:

Hanifi Enes Gül 115200085

Atakan Kaya 115200074

İbrahim Doğan 11512112

# Abstract

This Project is a Deep Q-Reinforcement Learning Project. the purpose of this project: without taking any human data, only actions that can be taken and the target specified model; to ensure that it reaches the target successfully. In this project, to ensure that the CartPole-v0 game selected from the OpenAI (GYM) library is successfully played in a way to achieve high scores. Simply we are building an autonomous agent that can successfully play the game.

# Introduction

There are three different sub-area of Machine Learning, which are Supervised Learning, Unsupervised Learning and Reinforcement Learning (RL). Reinforcement Learning technique will be implemented in this project. (Reinforcement Learning is the most academic paper written category in Machine Learning [2018]).

In this project, Reinforcement Learning methodology will be used with OpenAI Gym toolkit and Keras. Combination of RL and Neural Networks is called Deep Reinforcement Learning.

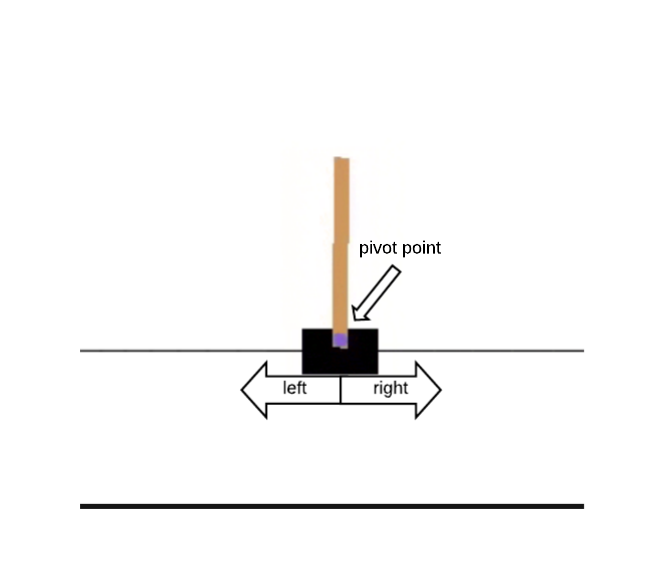
# Environment

We create an agent which performs “actions” in an “environment” and the agent receives various “rewards” depending on what “state” it is in when it performs the action.

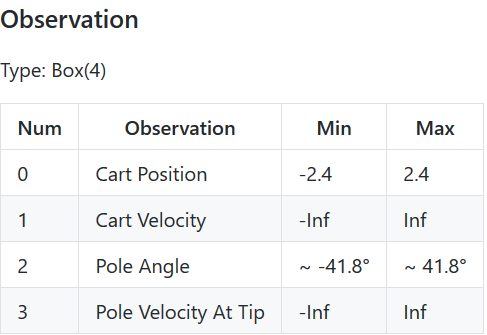
As it is already mentioned above, the game that selected from OpenAI Gym Toolkit is CartPole-v0. This game will be considered as our “environment”.

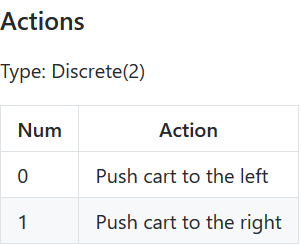


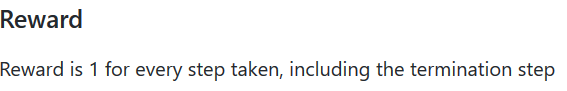
Environment (CartPole-v0) comes with its own rules and actions, observations etc. which are shown in next page in detail.



# Documentation of Environment

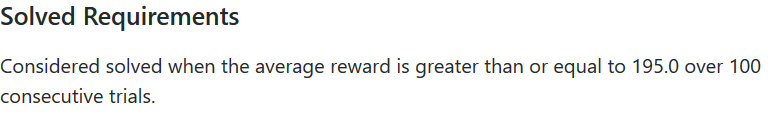






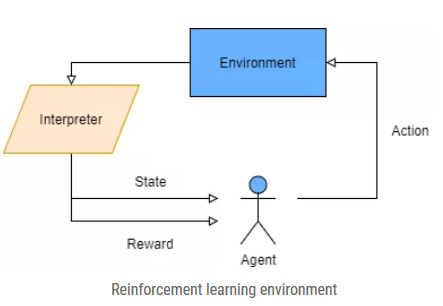
ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu



Resource: <https://github.com/openai/gym/wiki/CartPole-v0>

# Reinforcement Learning

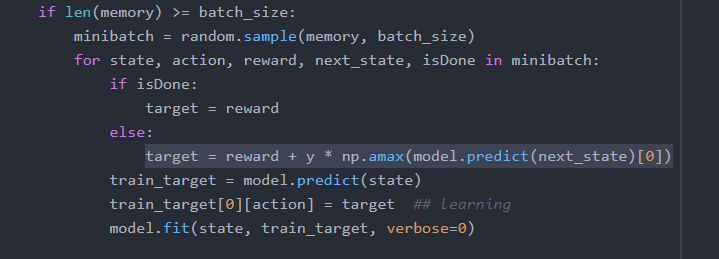
As can be seen below picture, the agent plays out some activity in the environment. A mediator sees this action in the environment, and feeds back a refreshed state that the agent currently lives in, and furthermore the reward for making this move. The environment isn't known by the agent previously, but instead it is found by the agent making steady strides in time. In this way, for example, at time t the agent, in state st, may make a move a. This outcomes in another state st+1 and a reward r. This reward can be a positive real number, zero, or a negative real number. (In this environment negative real number reward is not included.) It is the objective of the agent to realize which state subordinate action to take which boosts its rewards.

# Keras + Reinforcement Learning

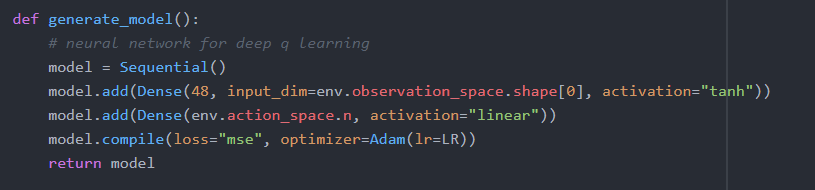
As it is mentioned, one of the aim is creating neural network that can perform Q-Learning. Current state is a must as an input. Q values will bi created for each action in that specific state with respect to Bellman Equation.



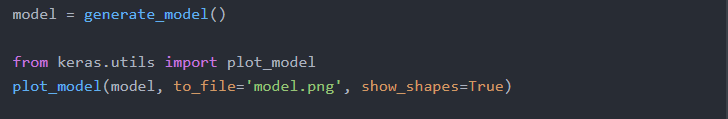
Our implementation is the highlighted part below:

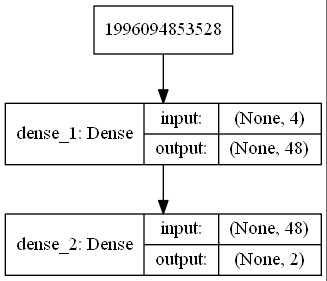


We build our model with this code block:



Keras has built-in model.png generator:



When we run this the output is:

To make it more understandable we illustrated it as:

# The main part:

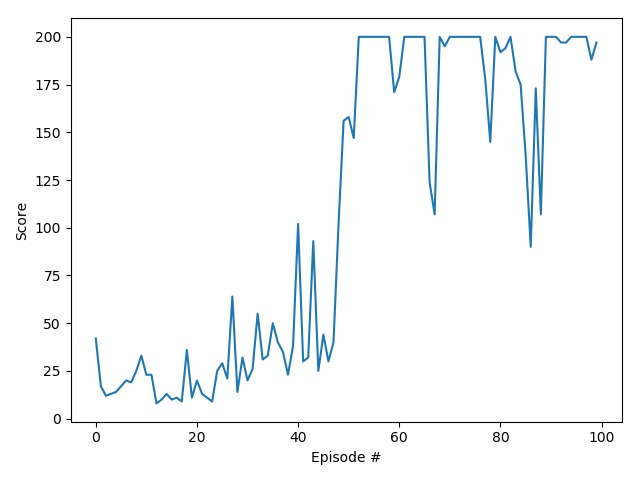
* Each episode runs until the environment sets the done=True.
* We keep score values to be able to see the success of episode.
* After applying epsilongreedy method we take step from environment and get reward.
* We add reward to score.
* We use memory to feed model regard to randomness with batchsize.
* Q-Learning part with bellman equation applied and fitted to model.
* Lastly, to check the accuracy of the agent we printed and plotted some values.

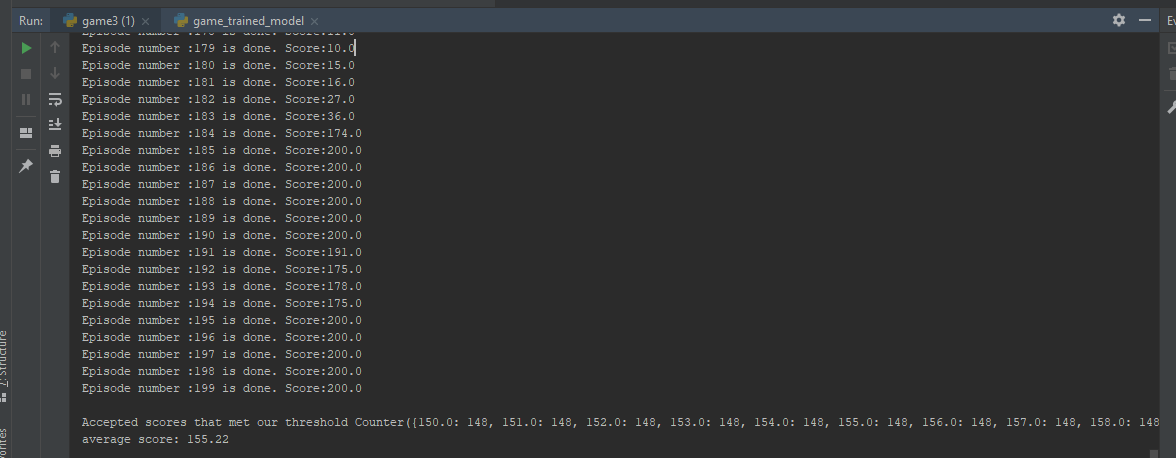
for episode in range(episodes):  
 s = env.reset()  
 s = np.reshape(s, [1, env.observation\_space.shape[0]])  
 score = 0  
  
 while True:  
  
 game\_memory = []  
 prev\_obs = []  
 env.render() #optional- to visualize  
 #epsilongreedy method  
 if random.uniform(0, 1) <= eps:  
 action = env.action\_space.sample()  
 else:  
 action = np.argmax(model.predict(s)[0])  
  
 if eps > eps\_min:  
 eps \*= eps\_decay  
  
 #taking step and getting necesssary informations (basics of RL)  
 new\_observation, reward, done, info = env.step(action)  
 new\_observation = np.reshape(new\_observation, [1, env.observation\_space.shape[0]])  
 # new\_observation=new\_observation.reshape(-1,len(new\_observation),1)  
  
 prev\_obs = new\_observation  
 score += reward  
  
 # In case we want to use another model  
 # we may want to use this one-hot encoding method,  
 # this is not currently used.  
  
 # training\_data = []  
 # if score >= score\_requirement:  
 # accepted\_scores.append(score)  
 # for data in game\_memory:  
 # # convert to one-hot (this is the output layer for our neural network)  
 # if data[1] == 1:  
 # output = [0, 1]  
 # elif data[1] == 0:  
 # output = [1, 0]  
 #  
 # # saving our training data  
 # training\_data.append([data[0], output])  
  
   
  
 memory.append((s, action, reward, new\_observation, done))  
  
 s = new\_observation  
 # env.render()  
 if len(memory) >= batch\_size:  
 minibatch = random.sample(memory, batch\_size)  
 for state, action, reward, next\_state, isDone in minibatch:  
 if isDone:  
 target = reward  
 else:  
 target = reward + y \* np.amax(model.predict(next\_state)[0])  
 train\_target = model.predict(state)  
 train\_target[0][action] = target ## learning  
 model.fit(state, train\_target, verbose=0)

scores\_window.append(score)  
  
 if score >= score\_requirement:  
 accepted\_scores.append(score)  
  
 if done:  
 print("Episode number :{} is done. Score:{}".format(episode, score))  
 break  
 scores.append(score)  
  
# just in case you wanted to reference later,  
# our training\_data that we already converted one-hot  
# training\_data\_save = np.array(training\_data)  
# np.save('saved.npy',training\_data\_save)  
  
fig = plt.figure()  
ax = fig.add\_subplot(111)  
plt.plot(np.arange(len(scores)), scores)  
plt.ylabel('Score')  
plt.xlabel('Episode #')  
plt.show()  
  
print("Accepted scores that met our threshold {}".format(Counter(accepted\_scores)))  
print('average score:', sum(scores) / len(scores))

# Plots and Prints

As we can see in the graph, when episode number increases our agent’s success also increased.





# Discussion

We tried to implement deep reinforcement learning using Keras to OpenAI’s Gym/CartPole-v0 environment. With doing some dynamic changes to code it can also work on other environments such as MountainCar-v0 etc. We commented one-hot encoding part of the code, with using correct sizes other neural networks can be also implemented such as TFLearn, DNN.

We get success in early steps nearly 40th episode. It can be improved, but it is quite successful.

# Appendix (All Code)

#necessary libraries  
import gym  
import random  
import numpy as np  
from collections import deque  
from collections import Counter  
from keras.models import Sequential  
from keras.layers import Dense  
from keras.optimizers import Adam  
import matplotlib.pyplot as plt  
  
#creating environment  
env = gym.make("CartPole-v0")  
env.reset()  
episodes = 100  
LR = 1e-3 #learningRate  
  
score\_requirement = 150 # our threshold  
scores = []  
accepted\_scores = [] # scores that met our threshold  
scores\_window = deque(maxlen=30)  
  
y = 0.95 #gamma  
eps = 0.5 #epsilon values for epsilon greedy method  
eps\_min = 0.01  
eps\_decay = 0.995  
decay\_factor = 0.999  
  
batch\_size = 16 #batch size  
  
memory = deque(maxlen=1000)  
  
print("State/Observation space:", env.observation\_space.shape)  
print("Number of actions:", env.action\_space)  
  
  
def generate\_model():  
 # neural network for deep q learning  
 model = Sequential()  
 model.add(Dense(48, input\_dim=env.observation\_space.shape[0], activation="tanh"))  
 model.add(Dense(env.action\_space.n, activation="linear"))  
 model.compile(loss="mse", optimizer=Adam(lr=LR))  
 return model  
  
  
model = generate\_model()  
  
for episode in range(episodes):  
 s = env.reset()  
 s = np.reshape(s, [1, env.observation\_space.shape[0]])  
 score = 0  
  
 while True:  
  
 game\_memory = []  
 prev\_obs = []  
 env.render() #optional- to visualize  
 #epsilongreedy method  
 if random.uniform(0, 1) <= eps:  
 action = env.action\_space.sample()  
 else:  
 action = np.argmax(model.predict(s)[0])  
  
 if eps > eps\_min:  
 eps \*= eps\_decay  
  
 #taking step and getting necesssary informations (basics of RL)  
 new\_observation, reward, done, info = env.step(action)  
 new\_observation = np.reshape(new\_observation, [1, env.observation\_space.shape[0]])  
 # new\_observation=new\_observation.reshape(-1,len(new\_observation),1)  
  
 prev\_obs = new\_observation  
 score += reward  
  
 # In case we want to use another model  
 # we may want to use this one-hot encoding method,  
 # this is not currently used.  
  
 # training\_data = []  
 # if score >= score\_requirement:  
 # accepted\_scores.append(score)  
 # for data in game\_memory:  
 # # convert to one-hot (this is the output layer for our neural network)  
 # if data[1] == 1:  
 # output = [0, 1]  
 # elif data[1] == 0:  
 # output = [1, 0]  
 #  
 # # saving our training data  
 # training\_data.append([data[0], output])  
  
 scores\_window.append(score)  
  
 if score >= score\_requirement:  
 accepted\_scores.append(score)  
  
 memory.append((s, action, reward, new\_observation, done))  
  
 s = new\_observation  
 # env.render()  
 if len(memory) >= batch\_size:  
 minibatch = random.sample(memory, batch\_size)  
 for state, action, reward, next\_state, isDone in minibatch:  
 if isDone:  
 target = reward  
 else:  
 target = reward + y \* np.amax(model.predict(next\_state)[0])  
 train\_target = model.predict(state)  
 train\_target[0][action] = target ## learning  
 model.fit(state, train\_target, verbose=0)  
  
 if done:  
 print("Episode number :{} is done. Score:{}".format(episode, score))  
 break  
 scores.append(score)  
  
# just in case you wanted to reference later,  
# our training\_data that we already converted one-hot  
# training\_data\_save = np.array(training\_data)  
# np.save('saved.npy',training\_data\_save)  
  
fig = plt.figure()  
ax = fig.add\_subplot(111)  
plt.plot(np.arange(len(scores)), scores)  
plt.ylabel('Score')  
plt.xlabel('Episode #')  
plt.show()  
  
print()  
print("Accepted scores that met our threshold {}".format(Counter(accepted\_scores)))  
print('average score:', sum(scores) / len(scores))