Implementing a Deep Q Network on an OpenAI Gym Acrobot Environment

New York University

KYS254@NYU.EDU

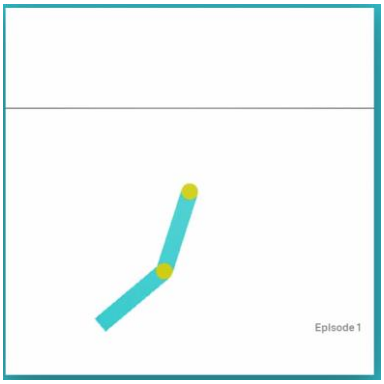
ME-GY 7973 New York University

New York University

Table of Contents

Project Overview

In this project I'm going to detail the process I undertook in building my Deep Q Network to drive an agent named "Acrobot-v1" from its rest state to a goal thus swinging the end-effector at a height at least the length of one link above the base. "Acrobot-v1" is from the OpenAI gym environment. The agent is described as an Acrobot system which includes two joints and two links, where the joint between the two links is actuated. Initially, the links are hanging downwards, and the goal is to swing the end of the lower link up to a given height. My learning network will be implemented in python through keras, a deep learning API which runs on the tensorflow platform. I decided to use keras solely based on the bias I have from having previous experience with it. A detailed a picture of the agent is provided below.



Problem Statement

In this environment, the problem is to get the double link Acrobot to swing up for its bottom link to reach the specified threshold shown by the black line above. A DQN has to be created to get the link to the threshold in the lowest number of moves. The more moves taken causes the agent to be penalized more and hence gets a lower reward. The goal is to get the highest reward possible by implementing a robust DQN network to train the Agent. In running the environment, a reward of -1 is given at every time step in which the goal is not achieved, therefore incentivizing speed of the Acrobot to maximize its rewards by completing the task quicker. The lowest reward attainable if the environment is not solved in the specified amount of time is -500

Environment Overview

To give a brief overview of the environment, every environment comes with an "action_space" and an "observation_space". The "observation_space" defines the structure of the observations your environment will be returning. Learning agents usually need to know this before they start running, in order to set up the policy function. The action_space used in the gym environment is used to define characteristics of the action space of the environment. With this, one can state whether the action space is continuous or discrete, define minimum and maximum values of the actions, etc. show plots or give what size observation and action Acrobot has and explain why it has that.

Code Breakdown

In my code generation process I house all my functions in one single class Deep_QN, mainly because I like the simplicity of having all my functions in one place and also for the fact that there were not that many functions for the need to create multiple class objects.

```
import gym  # open ai gym
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mp
import os
import time
import random
from keras.models import Sequential,load_model
from keras.layers import Dense, Dropout,Activation
from keras.optimizers import Adam
from collections import deque

%matplotlib notebook
```

This details all the imports I use in my code and their uses.

- Gym is the library that contains the environment I use for my Deep Q Network
- Numpy is used for scientific computation and handling the multidimensional arrays
- Matplotlib is used in creating plots of the finalized results output by my deep Q learning model
- Os is for accessing the weight files from the computer directory
- Time is used to calculate the run time of the code.
- The random library helps in obtaining random values
- Keras is the machine learning library containing the models, layers and optimizers used to build the deep learning model desired to train the Deep Q Network. "Sequential" is the first command needed to build the model and "load_model" is used to load

models previously saved. “Dense” is used to create layers, “Dropout” is used to randomly dropout a specified percentage of the weights in order to avoid memorization and force the model to learn and “Activation” is used to pass the activation functions. Lastly, “Adam” is an optimizer used in compiling the model.

- From the collections library, Deque is a double ended queue that has features of removing and adding elements from either end and is really convenient to use for memory storage of the rewards, states, etc,

```
def __init__(self,env,model_available,file_name):
    self.model_env = env
    self.gamma = 0.99
    self.epsilon = 1.0
    self.min_epsilon = 0.01
    self.dec_epsilon = 0.996
    self.learning_rate = 0.0005
    self.model_available = model_available
    self.weights_file = file_name
    self.batch_size = 32
    self.max_memory = 50_000
    self.memory = deque(maxlen=self.max_memory)
    if model_available:
        self.model = load_model(self.weights_file)
        self.model_target = load_model(self.weights_file)
    else:
        self.model = self.BuildModel()
        self.model_target = self.BuildModel()
```

The “init” function details the different hyperparameters being used for the Deep Q Network. [model_available] and [file_name] parameters are passed into the Class containing the name for the file for saved model weights and a Boolean on if there is an available saved model.

```
def BuildModel(self):
    model_input_shape = self.model_env.observation_space.shape       #model input sha
    model_output_shape = self.model_env.action_space.n              #model output sha
    model = Sequential()
    model.add(Dense(512, input_dim=model_input_shape[0],activation="relu"))
    model.add(Dense(512, activation="relu"))
    model.add(Dense(24, activation="relu"))
    model.add(Dense(self.model_env.action_space.n,activation="linear"))
    model.compile(loss="mean_squared_error",optimizer=Adam(lr=self.learning_rate))
    return model
```

My model is made of 4 layers. I use a 512 node for my input layer in addition with another dense layer after some experimentation. I use two activation function: “relu” and “linear” and use the “Adam” optimizer to compile my model.

```
def Recall(self, pres_state, action, reward, future_state, done):     #
    self.memory.append([pres_state,action,reward,future_state,done])

def ModelSave(self):      #save the model under specified name
    self.model.save(self.weights_file)       ##save model in specified
```

The “Recall” function is just used to store the present and future states, action, reward, and environment’s Boolean state through every step of the training process. The “ModelSave” function is also used to save the weights of the models under specific conditions mentioned in the main code, which also contains the location and name of the save file.

```
def Replay(self):
    if len(self.memory) < self.batch_size:           #if memory isn't up to bat
        return
    sample_minibatch = random.sample(self.memory,self.batch_size)       #sele
    for sm in sample_minibatch:                      #iterate through the sampl
        pres_state, action, reward, future_state, done = sm
        pres_state = pres_state[np.newaxis,:]         #reshape states
        future_state = future_state[np.newaxis,:]
        target_network = self.model_target.predict(pres_state)
        tn = self.model_target.predict(future_state)
        status = not done
        target_network[0,action] = reward + self.gamma*np.max(tn[0])*status
        self.model.fit(pres_state,target_network,epochs=1,verbose=0)
    self.epsilon = self.epsilon*self.dec_epsilon        #decay epsilon
    if self.epsilon < self.min_epsilon:     #if the epsilon value is past the m
        self.epsilon = self.min_epsilon
        return self.epsilon
```

The “Replay” function is the main function of a DQN network because it is where the model is trained. The model learns by sampling a batch of previous data from memory (mostly size 32 or 64) and iterates through the data updating the Q function. The Q function is expressed as:

$$y_t = g(x_t, u_t) + \alpha \max_a \hat{Q}(x_{t+1}, a, \theta^-)$$! here we use the target network

Here, we update the target Q function by adding the present reward to the possible future rewards (which is the expected future rewards multiplied my gamma) if done is not set to “True”. In my code, I do not initialize random weights for the Q network or target Q function but populate them through the first random iterations. I also reduce my epsilon value overtime in this function until epsilon equals the minimum set epsilon value and remains constant thereafter. The model fit line trains the target network

```
def ActionChoice(self,states):
    states = states[np.newaxis,:]            #reshape states by us
    rd = np.random.random()                 #select random value
    if rd < self.epsilon:
        action_select = self.model_env.action_space.sample()     #
    else:
        new_action = self.model.predict(states)           #predict
        action_select = np.argmax(new_action)              #select
    return action_select
```

My “ActionChoice” function is where I decide whether to take a random action based on my epsilon hyperparameter or continue the path I am already on and use past data. This function only takes in one input, states, and returns the index of the best action predicted by the model which is the maximum output of the array.

```
def T_network(self):
    model_theta = self.model.get_weights()
    self.model_target.set_weights(model_theta)
```

The function for updating the target network is simply extracting the current weights from the model and setting the weights into the target model.

Results

In this section, I am going to detail the different results I obtained as well as notable errors and how I went about them.

DQN Approach

My Deep Q Network approach was heavily influenced by class material and some additional online sources found through research. I experimented a couple of times with the target network after I discovered the versatility in how other researchers approached the solution. From soft target updates to double DQN I tried out different approaches until I settled on using a simple DQN due to better understanding. However, my implementation of the DQN run slower than usual causing incredibly long training hours. This was because I was running the REPLAY buffer at every time step. It however fun faster when I decided to run the REPLAY buffer outside of time step loop and in my episode loop. It worked a lot better and improved my run time and code performance. The equation I used in calculating my loss is seen below:

$$loss = \left(\underbrace{r + \gamma \max_{a`} \hat{Q}(s, a`)}_{\text{Target}} - \underbrace{Q(s, a)}_{\text{Prediction}} \right)^2$$

(Reward, Decay Rate)

Code Issues

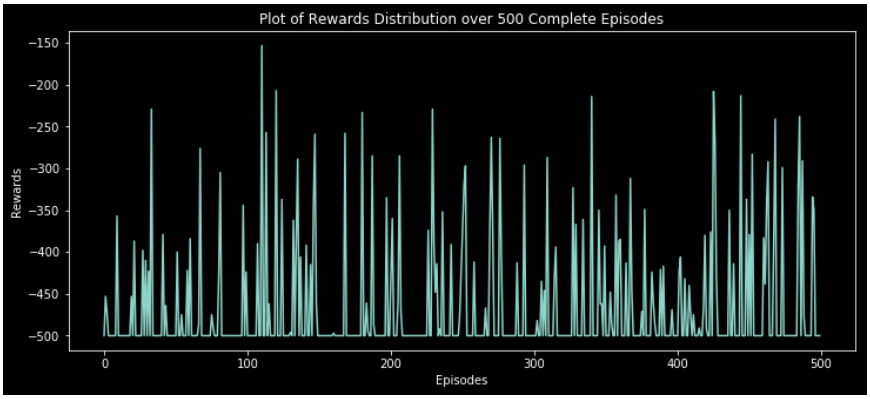
I tried out numerous combinations and run a lot of iterations changing hyper-parameters like learning rates, epsilon, batch size, memory size etc. but most of these iterations ended up not producing any good result at first. This can be partly attributed to my lack of patience and not letting the algorithm train for long hours because I wasn't sure if the initial rewards were constantly "-500" because there was a bug in the code or the network wasn't learning. This was a dilemma I faced for a while and caused me to make a lot of changes after every iteration and unfortunately not see patterns that would have given me better results. Also, I used a small memory size (2000) in my first iterations and used random deletions and FIFO methods to replace old memory with new memory when it got full. In hindsight, this did not help my code that much and made me loose a lot of relevant information needed to train the DQN model. I increased my memory size to about 50,000 for better results. Another thing I forgot to do was save my model weights for good models and reuse them later, which could have been useful in giving me better results.

Realizations

- Any Combination of layers I created almost always never gave me good results if it had a dropout layer and I am not sure if there is a valid reason or just bad coincidences. (Dropout layers delete a specified percentage of the weights to avoid memorization by the model)
- Saving weights and reloading them in your next run helps a lot if your model is learning properly.
- Another realization I noticed in running my DQN network was that more Dense layers did not necessarily mean better performance. This again was pretty surprising and could be just because of how my network is made but still surprised me nonetheless.

- Running a multiple 5000 iterations using the saved weights gives better results than running 1 full 50000 iteration. (Probably due to my low learning rate). This is evidenced below

Plots Progress and Descriptions



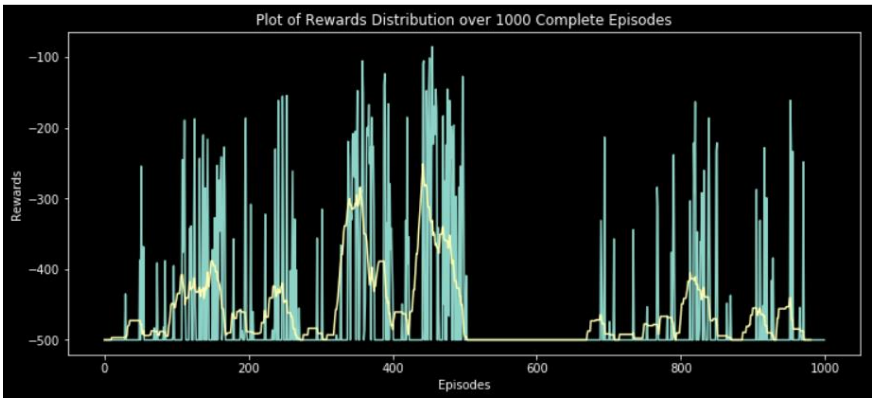
Number of episodes – 500

Learning rate – 0.01

Max memory – 50000

Batch size – 32

REPLAY buffer per 100 iterations



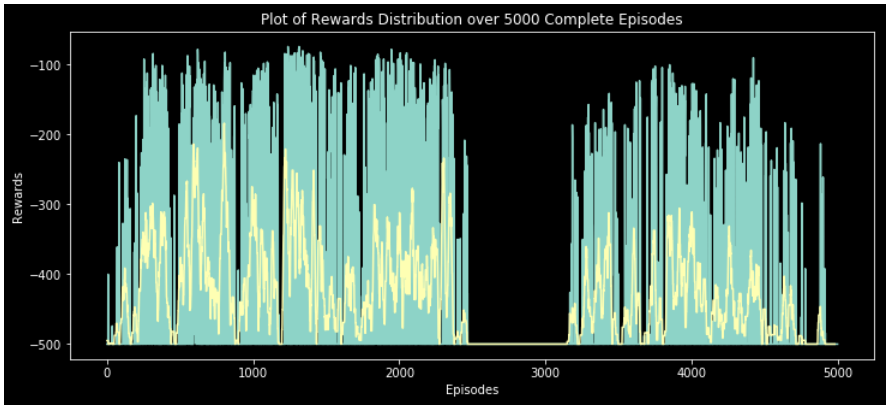
Number of episodes – 1000

Learning rate – 0.01

Max memory – 50000

Batch size – 32

REPLAY buffer per 100 iterations



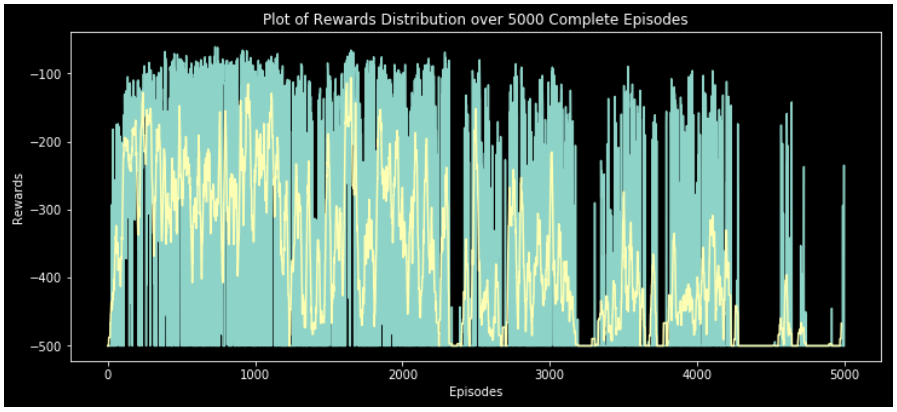
Number of episodes – 5000

Learning rate – 0.001

Batch size – 32

REPLAY buffer per 50 iterations

Three 24-node dense layers



Number of episodes – 5000

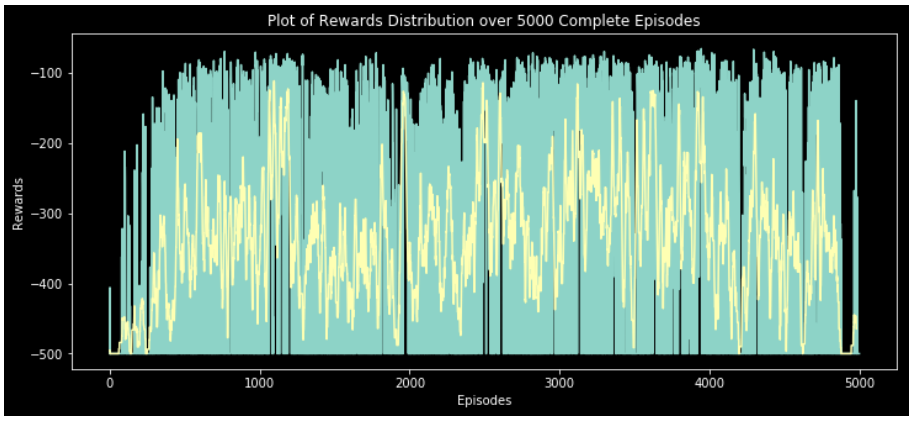
Learning rate – 0.001

Batch size – 32

REPLAY buffer per 50 iterations

Mean over total episodes – (-376.9228)

Three 24-node dense layers



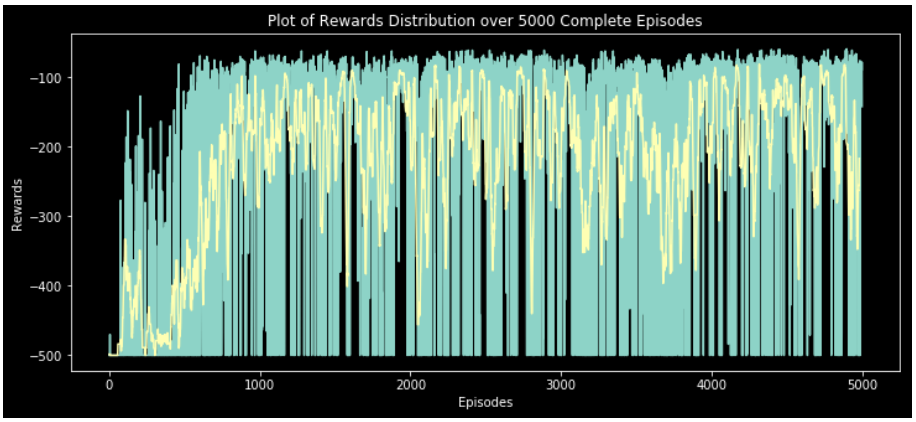
Number of episodes – 5000

Learning rate – 0.001

Batch size – 32

REPLAY buffer per 60 iterations

Mean over total episodes – (-340.1524)



Number of episodes – 5000

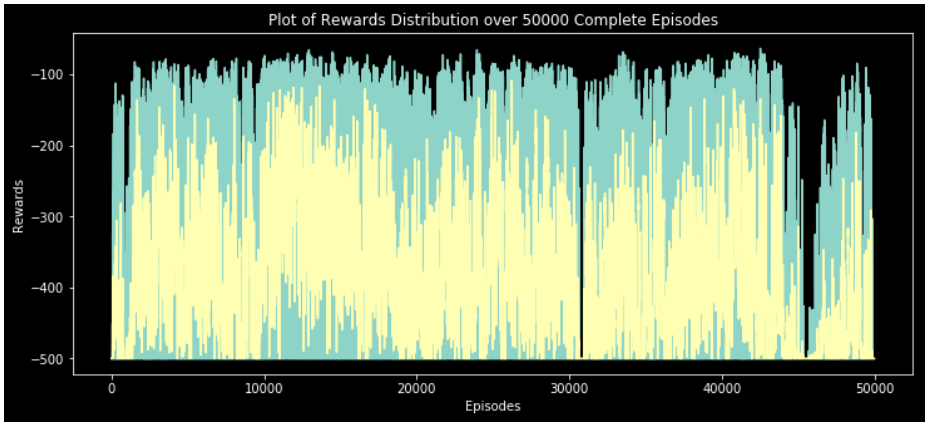
Learning rate – 0.001

Batch size – 32

REPLAY buffer per 60 iterations

Mean over total episodes – (-223.7)

Highest reward - -59



Number of episodes – 50000

Learning rate – 0.0005

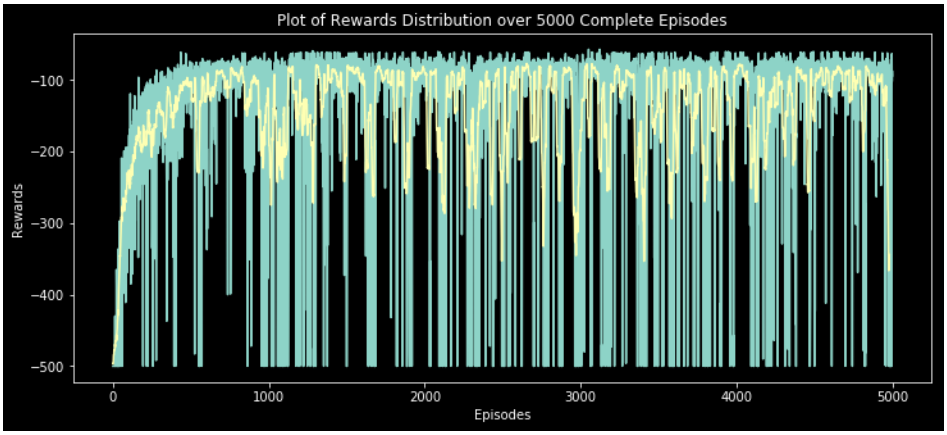
Batch size – 32

REPLAY buffer per 60 iterations

Mean over total episodes – (-380.7165)

Highest reward - -63

Best Result



Number of episodes – 5000

Learning rate – 0.0005

Batch size – 32

REPLAY buffer per 60 iterations

Mean over total episodes – (-143.48)

Highest reward - -57

Code Improvements

Even though my code was able to get the Acrobot agent to complete the task within a good amount of time the solution is still far from optimal. The best solution available completes the task with a reward of -42.37 ± 4.83 so there are still a lot of improvements to get to that level of efficiency. My plus-minus is also large as seen from my graphs and will need to get better. Experimenting with soft target updates and double DQNs might help greatly as well as a better dense layer structure for my model.

Conclusion

To conclude, this project was exciting as well as frustrating, but the knowledge gained was worth it. I run over 300,00 iterations combining different layer structures, changing hyperparameters and other code details. The Acrobot agent was a bit challenging but my initial struggle helped me understand the concept of DQNs more and ways to improve them. The final build model is shown below.

Relevant links

- Link to Acrobot – https://gym.openai.com/envs/Acrobot-v1/
- Link to Openai Gym – https://gym.openai.com/docs/#installation
- Link to keras – https://keras.io/