EL2805 Lab2 Report

Jiaying Yang jiayingy@kth.se

940816-0258 yipingx@kth.se

1 Problem 1: Deep Reinforcement Learning for Cartpole

1.1 a) Formulate the RL problem

State Space:

$$S = (P, V, \theta, V_{tip}),\ -2.4m \leq P \leq 2.4m,\ V, Vtip \in R,\ -41.8^\circ \leq \theta \leq 41.8^\circ$$

where P is cart position, V is cart velocity, θ is pole angle and V_{tip} is pole velocity at tip.

Action Space:

$$A = \{0, 1\}$$

where 0 indicates pushing cart to the left with 1N, and 1 indicates pushing cart to the right with 1N.

Termination:

When the episode number is 200 or the cart position or pole angle is out of the range.

Rewards:

$$R(-12.5^\circ \leq \theta \leq 12.5^\circ, a = \cdot) = 1$$
$$R(\theta \leq -12.5^\circ, a = \cdot) = 0$$
$$R(\theta \geq 12.5^\circ, a = \cdot) = 0$$

Q-learning works perfectly fine if we have a limited state space and action space. However, when the state space become much bigger, or even there are unlimited number of states like our problem, we need to have millions of records stored in a table in the program memory. This is definitely not realistic, thus we use neural network to estimate Q function and calculate the Q values.

1.2 b) Brief Description

The association of line numbers in code with lines in pseudocode is in the Appendix.

In the main function, first we initialize the CartPole-v0 environment, DQN agent and the arrays to collect test states for plotting Q values using random policy.
Then for every episode, we initialize the state first. And then get action for current state based on ϵ-greedy policy and go one step in environment, train the NN model, collect reward and store them and propagate the state, if it is not at the end of the episode. If it is at the end episode, we just update the target network and store the scores for plotting. If the check_solve flag is True, we calculate the mean of scores of last 100 episodes to check if it is larger than 195. If the average score is larger than 195, we stop the training process and mark as solved, otherwise we continue training.
The behavior of each function is:
__ init__: the initialization of global variables.
build_model: use Keras to build the neural network model.
update_target_model: update target network model parameters, and make it the same with the model.
get_action: the implementation of ϵ-greedy policy, which is to determine how to select actions.

append_sample: save sample {s,a,r,s'} to the replay memory.
train_model: implement the process of DQN.
plot_data: draw the image of average Q value and score.

1.3 c) Brief Explanation of Layout of NN

The layout of given Neural Network has two layers. The first layer has 16 neurons with nonlinear activation 'Relu', and the second layer has 2 neurons with a linear activation. Both layers use He Initialization to increase the stability and fasten the gradient descent algorithm.

1.4 d) $get\text{-}action$ function

Please see the code in Appendix.

1.5 e) $train\text{-}model$ function

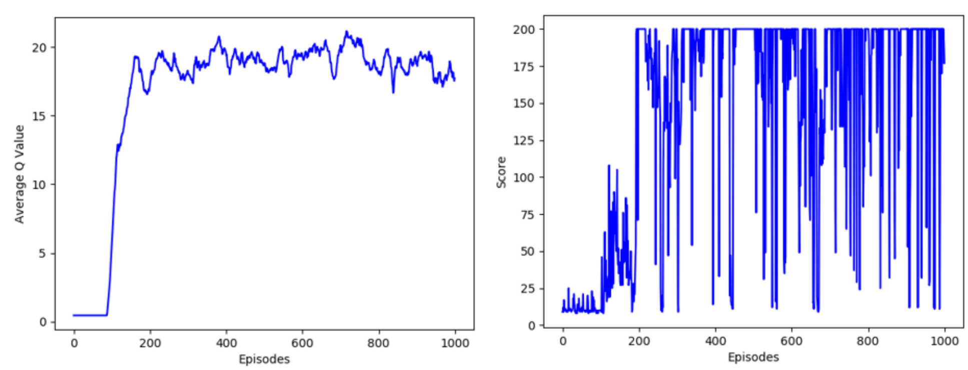
Please see the code in Appendix.

1.6 f) Complexity of neural network

To compare the influence of the complexity of neural network, we first set the other hyper parameters to be fixed. Their values are:

$$
\begin{aligned}
&discount\ factor = 0.95\\
&learning\ rate = 0.005\\
&epsilon = 0.02\\
&batch\ size = 32\\
&memory\ size = 1000\\
&train\ start = 1000\\
&target\ update\ frequency = 1
\end{aligned}
$$

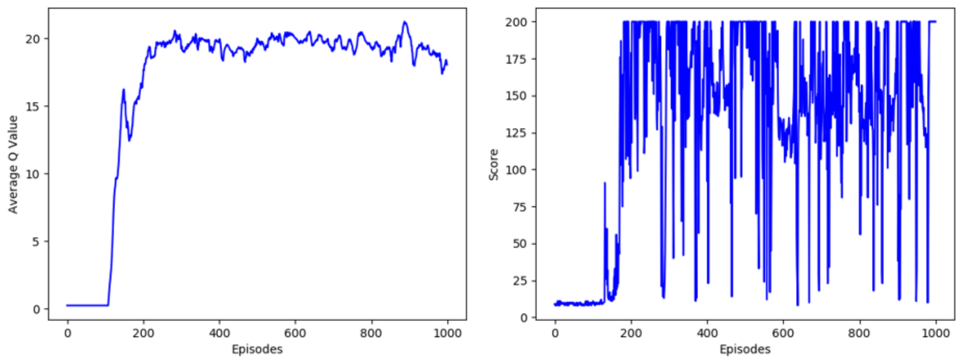
What is more, for each layer, the activation function is set as 'Relu' all the time. We change the structure of neural network for many times to evaluate the networks. Our result is shown from Figure 1 to Figure 5:



Network structure: one hidden layer – 16 neurons

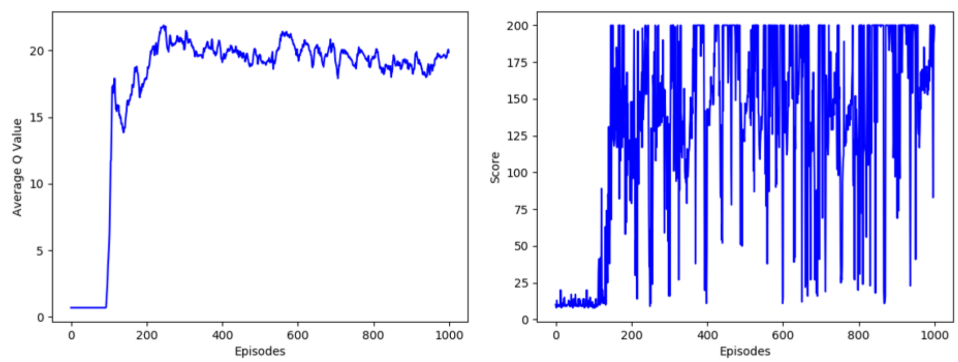
Figure 1: Network structure complexity case 1

As we can note, the agent learns faster with more nodes and more layers. From Figure 1 to Figure 5, the agent takes fewer steps to get high scores with more neurons since more layers and more nodes in the layer, we increase the complexity of the model, which has a better approximation of our problem. Thus we find the model with 32 neurons in the first layer and 32 neurons in the second layer the best and we will keep using this model in the following questions.



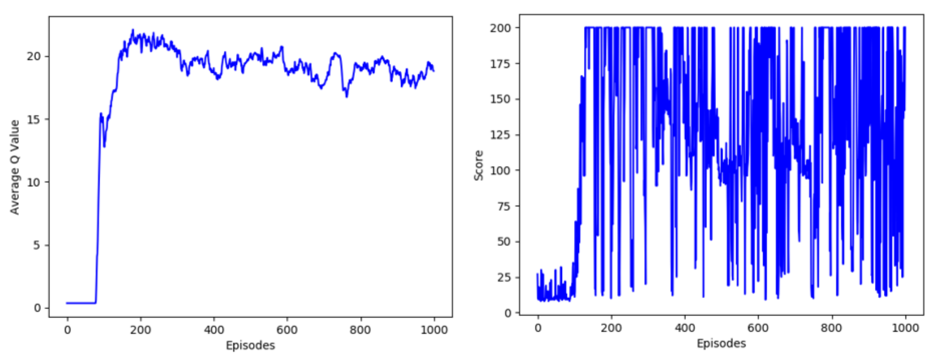
Network structure: one hidden layer – 32 neurons

Figure 2: Network structure complexity case 2



Network structure: two hidden layers – 16 neurons for the 1^{st} layer, and 16 neurons for the 2^{nd} layer

Figure 3: Network structure complexity case 3



Network structure: two hidden layers – 16 neurons for the 1^{st} layer, and 32 neurons for the 2^{nd} layer

Figure 4: Network structure complexity case 4

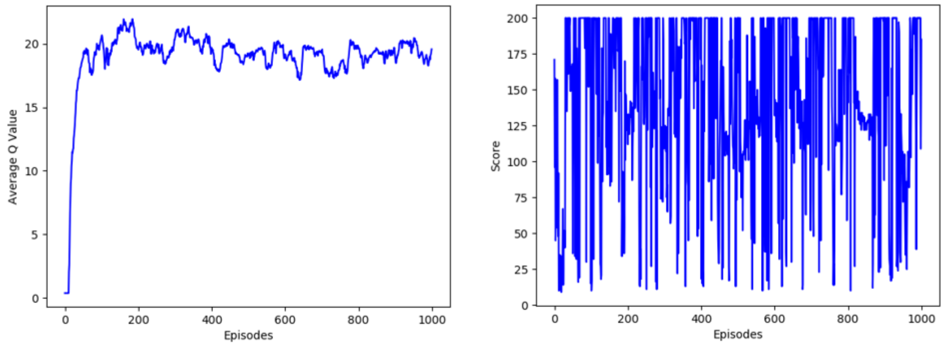
1.7 g) Effect of discount factor, learning rate, memory size

Discount factor

As we can observe from Figure 6 to Figure 8, the bigger discount factor is, the larger averaged Q value the agent can get, and the slower the average Q value will converge at the same time. Since small discount factor will force the agent focus on the current reward more, which is a disadvantage. So we decide to use discount factor 0.995 in the following questions.

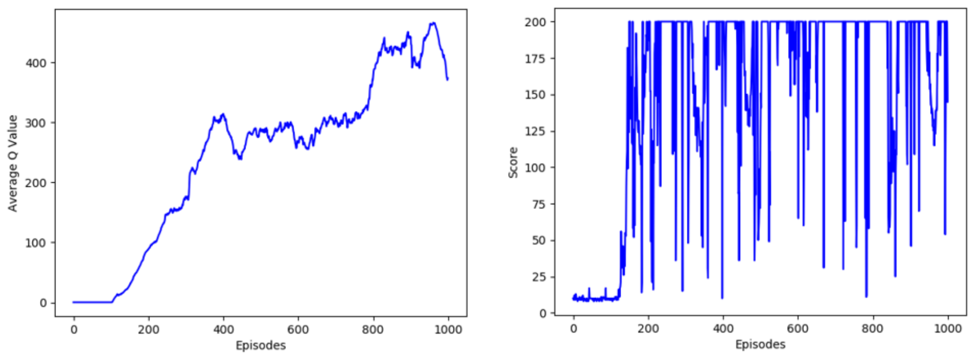
Learning rate

As we can observe from Figure 9 to 11, a larger learning rate makes the agent learns faster but a



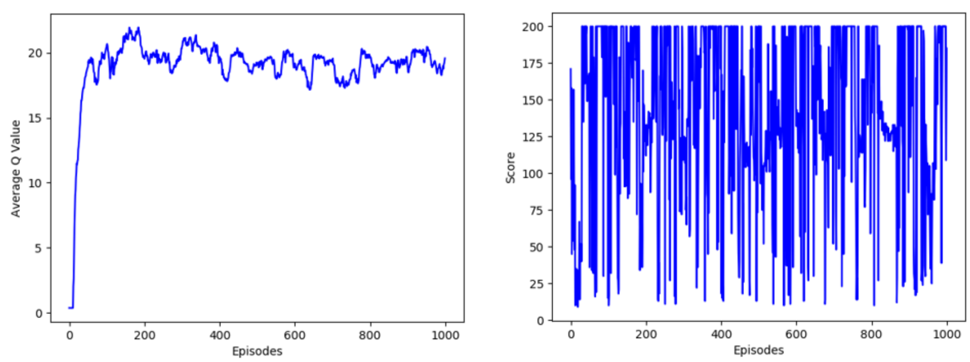
Network structure: two hidden layers – 32 neurons for the 1^{st} layer, and 32 neurons for the 2^{nd} layer

Figure 5: Network structure complexity case 5



Discount factor: 0.995

Figure 6: Discount factor case 1



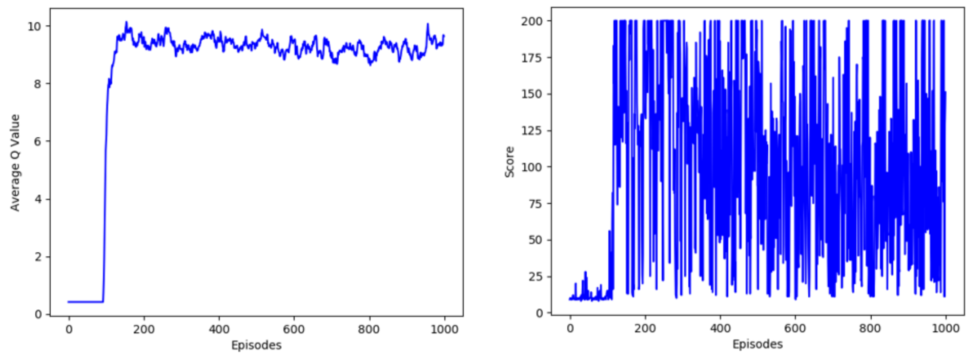
Discount factor: 0.95

Figure 7: discount factor case 2

smaller learning rate makes the agent learns more stable. In order to have a relatively stable and fast learning process, we should choose the learning rate neither too large nor too small. Thus we decide to choose 0.005 as learning rate for following questions.

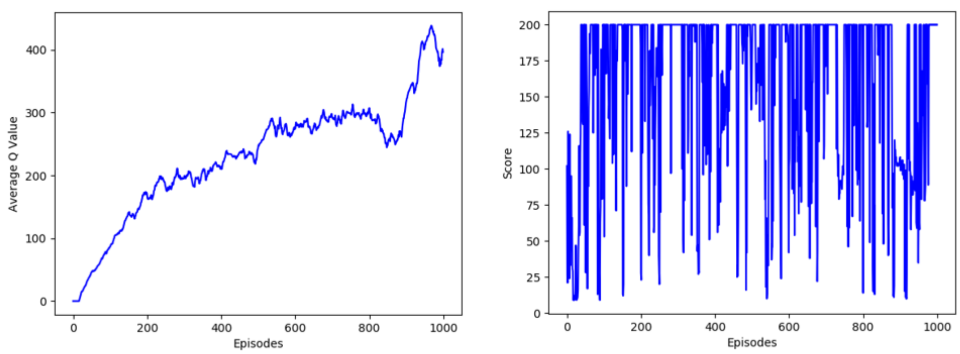
Memory size

As we can observe from Figure 12 to 14, when memory size is too small, namely smaller than 1000, the training process will not work. With larger memory size, since the agent can use more samples for training, it gives us better performance, but at the same time we observed that it took longer time to train the model. We will use memory size 10000 for the following questions.



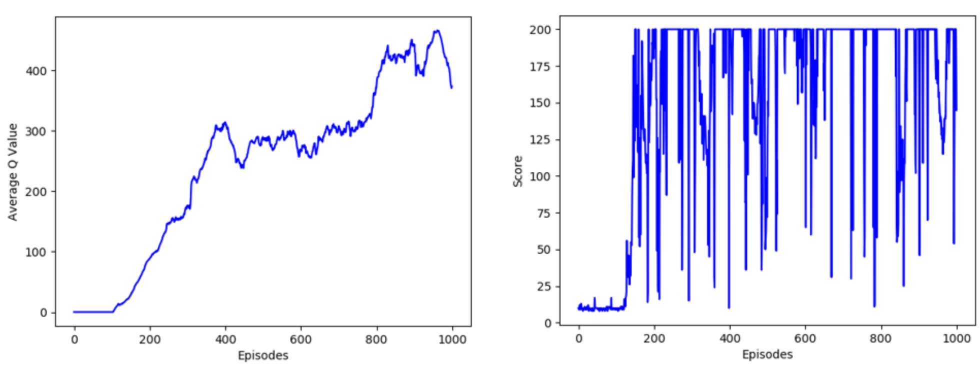
Discount factor: 0.9

Figure 8: discount factor case 3



Learning rate: 0.01

Figure 9: Learning rate case 1

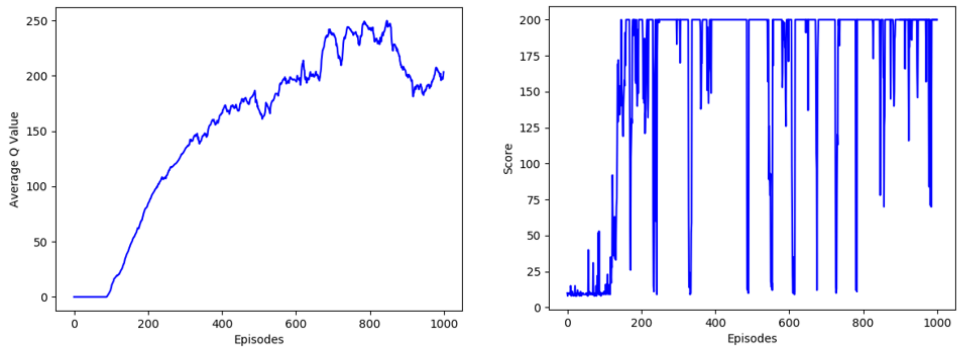


Learning rate: 0.005

Figure 10: Learning rate case 2

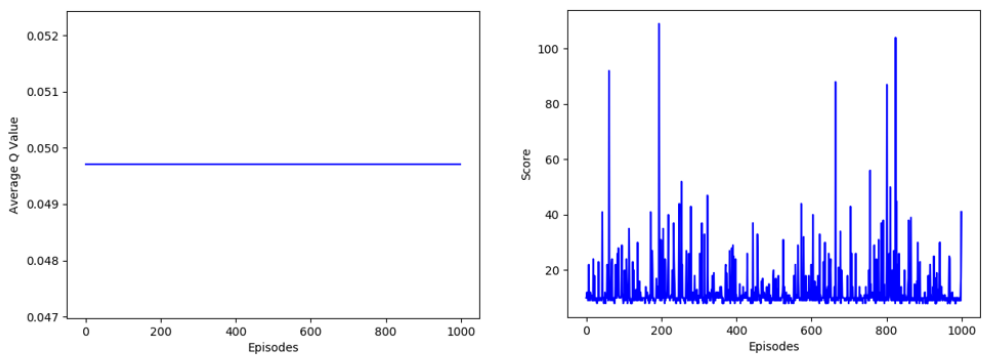
1.8 h) Effect of Target Update Frequency

As we can observe from Figure 15 to 18, target update frequency 1 gives us the best performance. The update frequency means how often we update the network, and a larger one will make the agent learn slower. From what we observe, frequency 1 gives the best performance. So we will use all improved hyper parameters to check if the problem solved.



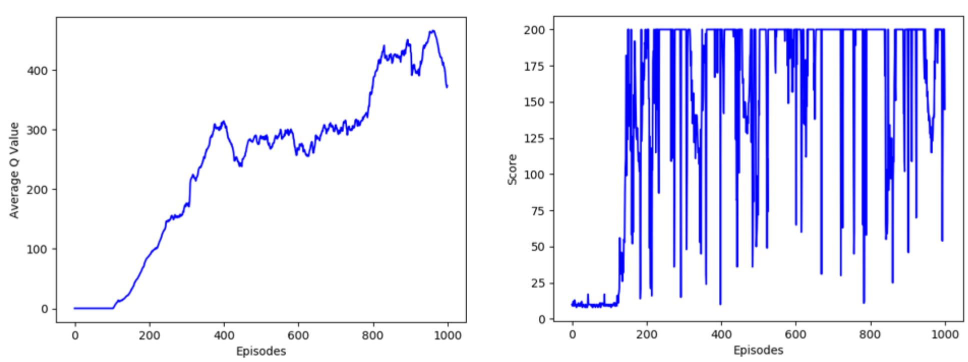
Learning rate: 0.002

Figure 11: Learning rate case 3



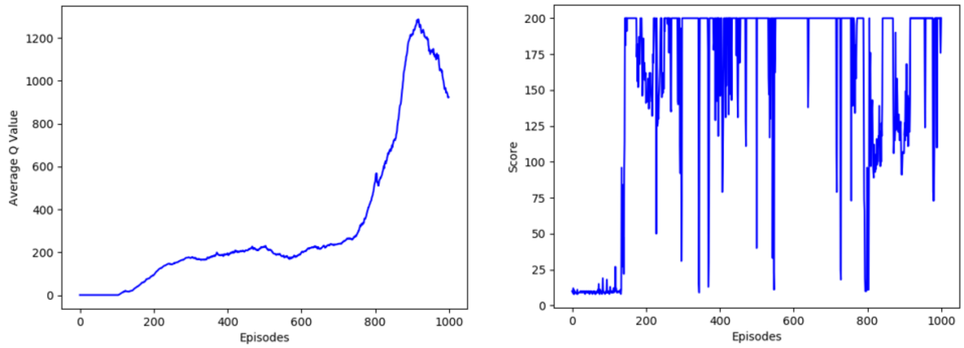
Memory size: 500

Figure 12: Memory size case 1



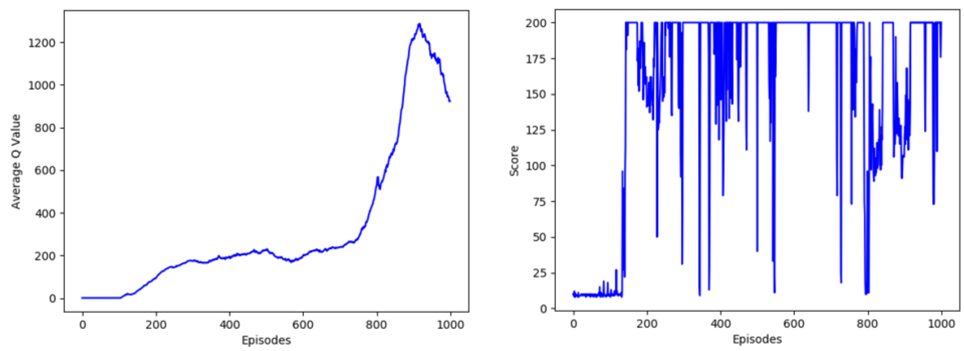
Memory size: 1000

Figure 13: Memory size case 2



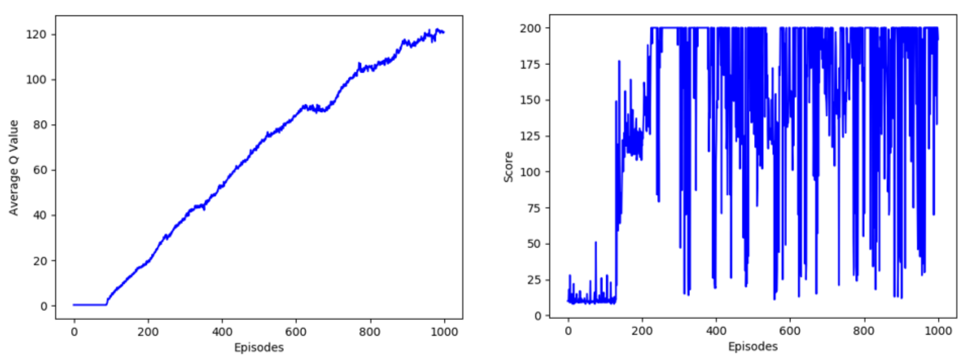
Memory size: 10000

Figure 14: Memory size case 3



Target update frequency: 1

Figure 15: Target update frequency case 1



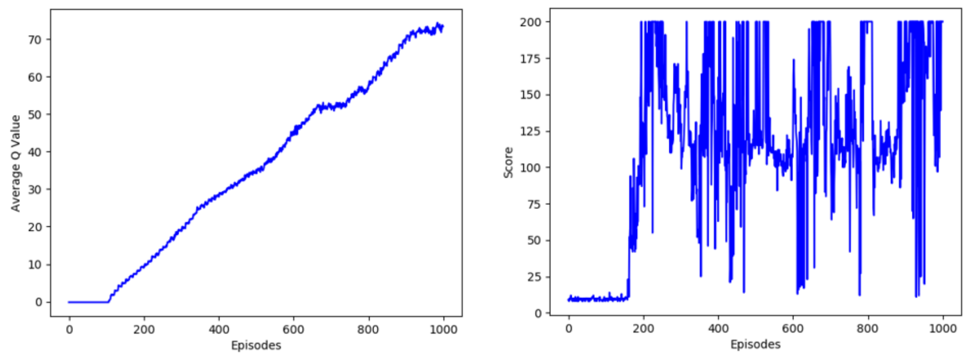
Target update frequency: 5

Figure 16: Target update frequency case 2

1.9 i) Performance of an Appropriate Set of Hyperparameters

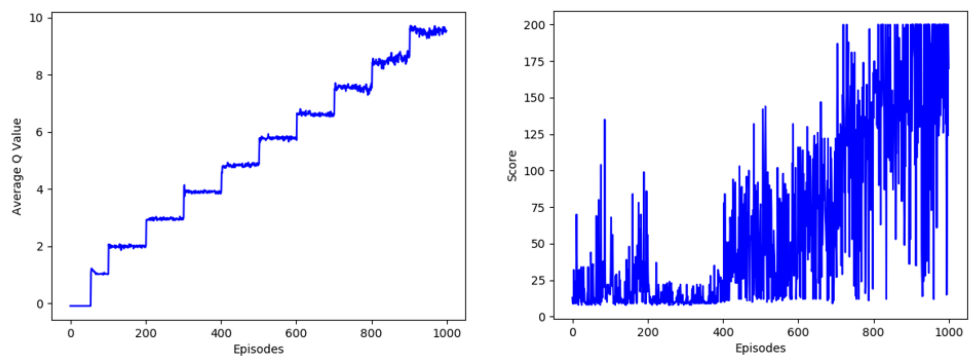
Finally we choose:

$$
\begin{aligned}
&discount\ factor = 0.995 \\
&learning\ rate = 0.005 \\
&epsilon = 0.02 \\
&batch\ size = 32 \\
&memory\ size = 10000 \\
&train\ start = 1000 \\
&target\ update\ frequency = 1
\end{aligned}
$$



Target update frequency: 10

Figure 17: Target update frequency case 3



Target update frequency: 100

Figure 18: Target update frequency case 4

And observe that the problem solved after episode 724. The Q value and score plot can be found in Figure 19.

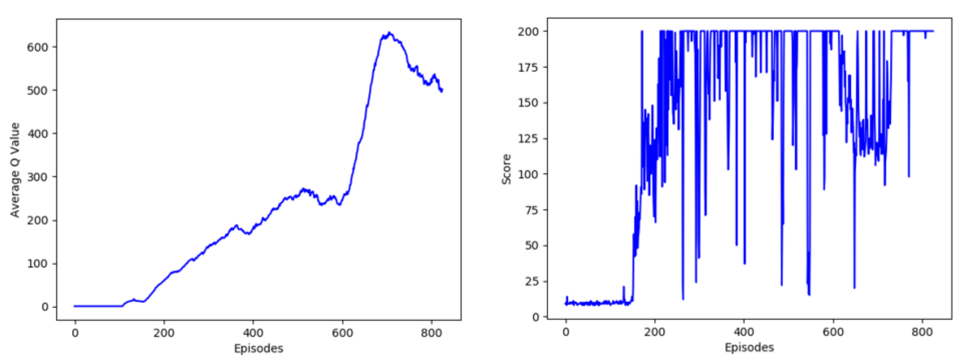


Figure 19: Performance of an Appropriate Set of Hyperparameters

2 Appendix

Line 203-215 in code is to initialize the first state and compute q-values, which corresponds to pseudocode line 1.

Line 216-219 in code is to do the iteration, which corresponds to pseudocode line 2.

Line 220-227 in code is to use ϵ-greedy policy to get action and update the next state, which corresponds to pseudocode line 3-5.

Line 229-231 in code is to train the agent, which corresponds to pseudocode line 6-12.

Line 233-242 in code is to update target network parameters 'frequency', which corresponds to pseudocode line 14.

Line 244-250 in code is to check if we can consider the problem is solved, by calculating the mean of scores of last 100 episodes, which corresponds to pseudocode line 15-17.

```
1  import sys
2  import gym
3  import pylab
4  import random
5  import numpy as np
6  from collections import deque
7  from keras.layers import Dense
8  from keras.optimizers import Adam
9  from keras.models import Sequential
10
11
12 EPISODES = 1000 #Maximum number of episodes
13
14 #DQN Agent for the Cartpole
15 #Q function approximation with NN, experience replay, and target network
16 class DQNAgent:
17     #Constructor for the agent (invoked when DQN is first called in main)
18     def __init__(self, state_size, action_size):
19         self.check_solve = True #If True, stop if you satisfy solution confition
20         self.render = False
21         #If you want to see Cartpole learning, then change to True
22
23         #Get size of state and action
24         self.state_size = state_size
25         self.action_size = action_size
26
27 ################################################################################
28 ################################################################################
29         #Set hyper parameters for the DQN. Do not adjust those labeled as Fixed.
30         self.discount_factor = 0.995
31         self.learning_rate = 0.002
32         self.epsilon = 0.02 #Fixed
33         self.batch_size = 32 #Fixed
34         self.memory_size = 10000
35         self.train_start = 1000 #Fixed
36         self.target_update_frequency = 1
37 ################################################################################
38 ################################################################################
39
40         #Number of test states for Q value plots
41         self.test_state_no = 10000
42
43         #Create memory buffer using deque
44         self.memory = deque(maxlen=self.memory_size)
45
46         #Create main network and target network (using build_model defined below)
47         self.model = self.build_model()
48         self.target_model = self.build_model()
49
50         #Initialize target network
```

```
        self.update_target_model()

    #Approximate Q function using Neural Network
    #State is the input and the Q Values are the output.
###############################################################################
###############################################################################
        #Edit the Neural Network model here
        #Tip: Consult https://keras.io/getting-started/sequential-model-guide/
    def build_model(self):
        model = Sequential()
        model.add(Dense(32, input_dim=self.state_size, activation='relu',
                        kernel_initializer='he_uniform'))
        model.add(Dense(32,  activation='relu',
                        kernel_initializer='he_uniform'))
        model.add(Dense(self.action_size, activation='linear',
                        kernel_initializer='he_uniform'))
        # model.add(Dense(16, input_dim=self.state_size, activation='relu',
        #                 kernel_initializer='he_uniform'))
        # model.add(Dense(self.action_size, activation='linear',
        #                 kernel_initializer='he_uniform'))
        model.summary()
        model.compile(loss='mse', optimizer=Adam(lr=self.learning_rate))
        return model
###############################################################################
###############################################################################

    #After some time interval update the target model to be same with model
    def update_target_model(self):
        self.target_model.set_weights(self.model.get_weights())

    #Get action from model using epsilon-greedy policy
    def get_action(self, state):
###############################################################################
###############################################################################
        #Insert your e-greedy policy code here
        #Tip 1: Use the random package to generate a random action.
        #Tip 2: Use keras.model.predict() to compute Q-values from the state.
        # action = random.randrange(self.action_size)
        state_actions = self.model.predict(state) # All actions in this state
        if (np.random.uniform() < self.epsilon) or (
                state_actions.all() == 0):
            # non-greedy or when this state has not been discovered
        # if (np.random.uniform() <= self.epsilon):
            action = random.randrange(self.action_size)
        else:
            action = state_actions.argmax()    # greedy
        return action
###############################################################################
###############################################################################
    #Save sample <s,a,r,s'> to the replay memory
    def append_sample(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))
        #Add sample to the end of the list

    #Sample <s,a,r,s'> from replay memory
    def train_model(self):
        if len(self.memory) < self.train_start: #Do not train if not enough memory
            return
        batch_size = min(self.batch_size, len(self.memory))
```

```
        #Train on at most as many samples as you have in memory
        mini_batch = random.sample(self.memory, batch_size)
        #Uniformly sample the memory buffer
        #Preallocate network and target network input matrices.
        update_input = np.zeros((batch_size, self.state_size))
        #batch_size by state_size two-dimensional array (not matrix!)
        update_target = np.zeros((batch_size, self.state_size))
        #Same as above, but used for the target network
        action, reward, done = [], [], [] #Empty arrays that will grow dynamically

        for i in range(self.batch_size):
            update_input[i] = mini_batch[i][0]
            #Allocate s(i) to the network input array from iteration i in the batch
            action.append(mini_batch[i][1]) #Store a(i)
            reward.append(mini_batch[i][2]) #Store r(i)
            update_target[i] = mini_batch[i][3]
            #Allocate s'(i) for the target network array
            # from iteration i in the batch
            done.append(mini_batch[i][4])  #Store done(i)

        target = self.model.predict(update_input)
        #Generate target values for training the inner loop network
        # using the network model
        target_val = self.target_model.predict(update_target)
        #Generate the target values for training the outer loop target network

        #Q Learning: get maximum Q value at s' from target network
###############################################################################
###############################################################################
        #Insert your Q-learning code here
        #Tip 1: Observe that the Q-values are stored in the variable target
        #Tip 2: What is the Q-value of the action taken
        # at the last state of the episode?
        # for i in range(self.batch_size): #For every batch
        #     target[i][action[i]] = random.randint(0,1)
        for i in range(self.batch_size):
            if done[i]:
                target[i][action[i]] = reward[i]
            else:
                target[i][action[i]] = reward[i] + \
                    self.discount_factor * np.max(target_val[i])
###############################################################################
###############################################################################

        #Train the inner loop network
        self.model.fit(update_input, target, batch_size=self.batch_size,
                       epochs=1, verbose=0)
        return
    #Plots the score per episode as well as the maximum q value per episode,
    # averaged over precollected states.
    def plot_data(self, episodes, scores, max_q_mean):
        pylab.figure(0)
        pylab.plot(episodes, max_q_mean, 'b')
        pylab.xlabel("Episodes")
        pylab.ylabel("Average Q Value")
        pylab.savefig("qvalues.png")

        pylab.figure(1)
        pylab.plot(episodes, scores, 'b')
```

```
        pylab.xlabel("Episodes")
        pylab.ylabel("Score")
        pylab.savefig("scores.png")

if __name__ == "__main__":
    #For CartPole-v0, maximum episode length is 200
    env = gym.make('CartPole-v0')
    #Generate Cartpole-v0 environment object from the gym library
    #Get state and action sizes from the environment
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n

    #Create agent, see the DQNAgent __init__ method for details
    agent = DQNAgent(state_size, action_size)

    #Collect test states for plotting Q values using uniform random policy
    test_states = np.zeros((agent.test_state_no, state_size))
    max_q = np.zeros((EPISODES, agent.test_state_no))
    max_q_mean = np.zeros((EPISODES,1))

    done = True
    for i in range(agent.test_state_no):
        if done:
            done = False
            state = env.reset()
            state = np.reshape(state, [1, state_size])
            test_states[i] = state
        else:
            action = random.randrange(action_size)
            next_state, reward, done, info = env.step(action)
            next_state = np.reshape(next_state, [1, state_size])
            test_states[i] = state
            state = next_state

    scores, episodes = [], [] #Create dynamically growing score and episode counters
    for e in range(EPISODES):
        done = False
        score = 0
        state = env.reset() #Initialize/reset the environment
        state = np.reshape(state, [1, state_size])
        #Reshape state so that to a 1 by state_size two-dimensional array
        # ie. [x_1,x_2] to [[x_1,x_2]]
        #Compute Q values for plotting
        tmp = agent.model.predict(test_states)
        max_q[e][:] = np.max(tmp, axis=1)
        max_q_mean[e] = np.mean(max_q[e][:])

        while not done:
            if agent.render:
                env.render() #Show cartpole animation

            #Get action for the current state and go one step in environment
            action = agent.get_action(state)
            next_state, reward, done, info = env.step(action)
            next_state = np.reshape(next_state, [1, state_size])
            #Reshape next_state similarly to state

            #Save sample <s, a, r, s'> to the replay memory
            agent.append_sample(state, action, reward, next_state, done)
```

```python
                #Training step
                agent.train_model()
                score += reward #Store episodic reward
                state = next_state #Propagate state

                if done:
                    #At the end of very episode, update the target network
                    if e % agent.target_update_frequency == 0:
                        agent.update_target_model()
                    #Plot the play time for every episode
                    scores.append(score)
                    episodes.append(e)

                    print("episode:", e, "  score:", score," q_value:", max_q_mean[e],
                          "  memory length:",len(agent.memory))

                    # if the mean of scores of last 100 episodes is bigger than 195
                    # stop training
                    if agent.check_solve:
                        if np.mean(scores[-min(100, len(scores)):]) >= 195:
                            print("solved after", e-100, "episodes")
                            agent.plot_data(episodes,scores,max_q_mean[:e+1])
                            sys.exit()

    pylab.figure()
    pylab.plot(episodes[-min(100, len(scores)):],
                scores[-min(100, len(scores)):], 'b')
    pylab.xlabel("Episodes")
    pylab.ylabel("Average Score in last 100 episode")
    pylab.savefig("AverageScoresLast100Episode.png")
    print( np.mean(scores[-min(100, len(scores)):]))
    agent.plot_data(episodes,scores,max_q_mean)
```