EL2805 Reinforcement Learning

Lab 2

Deep Q Learning

Pratima Rao A., Abgeiba Isunza Navarro

1 Problem (a)

Formulating the problem:
State space
The state space $\mathscr{S}$, comprising al possible frames can be represented using four parameters - the velocity of the cart (v), the position of the cart (p), the angle of the pole (θ) and pole velocity at the tip(vel).

$$\mathscr{S} = \{(v, p, \theta, vel) : v \in R, -2.4 \leq p \leq 2.4, -41.8^\circ \leq \theta \leq 41.8^\circ, vel \in R\}$$

Actions
There are two actions possible at each state:

$$\mathscr{A} = \{\text{left, right}\}$$

where these indicate moving the cart to the left and right respectively.
Rewards
The rewards for this are as follows:

- $((v, p, \theta, vel), .) = 1$ when $-12.5^\circ \leq \theta \leq 12.5^\circ$
- $((v, p, \theta, vel), .) = 1$ otherwise

Terminal state
The episode terminates after 200 time steps or once the agent receives 0 reward for the first time.
One problem with Q-learning is the lack of generalisation ability i.e. Q-learning agent does not have the ability to estimate value for unseen states. Additionally, state space is huge. Since every small change in any of the four parameters used in defining a state represents a new state, we would need to have a very big memory to store all possible states. This problem is overcome by using a DQN. In DQN the two-dimensional array used in Q-learning is replaced by introducing Neural Network.

2 Problem (b)

Here we provide a brief explanation of the functions found in *cartpole_dqn.py*

- ***main*:** In this function the environment from the CarPole is created and initialized. A DQN agent is created and the vectors from the test state cases are initialized with a random policy. The function then loops over N episodes, in each of them the environment is started each time. The mean of the maximum values of the Q vector are saved for plotting. In each episode and at each time step the agent will choose an action according to an ϵ-greedy policy, the environment will return the corresponding observation, reward and a variable indicating if it is time to reset the environment again; according to the information from the environment, the agent will save the states, actions and rewards which will be use to train the model (NN). A *score* variable is saved, which includes the reward for each episode. This scores will be plotted at the end of the N episodes.
- ***DQNAgent Class*:** This is a class of the DQN agent where the model and its parameters are initialized and saved. This class contains a function to determine the action of the agent, save the state, actions and Q values.
- ***build_model*:** This function is part of the agent class. It is dedicated to initialize and construct the NN model using Keras.
- ***update_target_model*:** This function is part of the agent class. It updates the target model to have the same weights than the current model.
- ***get_action*:** This function is part of the agent class. It implements the ϵ-greedy policy, and returns the action selected accordingly.
- ***append_sample*:** This function is part of the agent class. It saves a sample of (s,a,r,s') to the replay memory.
- ***train_model*:** This function is part of the agent class. It creates batches from the saved samples (s,a,r,s'). Then it obtains their Q values and trains the NN model.
- ***plot_data*:** This is a function to plot the scores and Q average values for each episode.

3 Problem (c)

The pseusocode given in 1 outlines the steps involved in DQN.

- The first three steps are the initialisation required for the model. These are done upon calling the class *DQNAgent* in the code.

- The first *for* loop indicates a loop over the number of episodes, i.e. the number of times the game is played. The objective is that as the agent plays more games, it begins to learn the optimal moves. This corresponds to line 184 in the code, and the initialization to the following lines.
- The next line represents the number of iteration which is implemented by the line *while not done* in the code.
- The following two lines are the code for the ϵ greedy policy which is used for picking the action. This is done by the function *get_action*.
- After this, the obtained action is executed (by calling *env.step(action)*) and the next state, reward are stored using *agent.append_sample*.
- The next few lines correspond to *agent.train_model()* in the code. Here, the minibatches are smapled uniformaly, and the model is trained by minimizing the diffrence between the values generated by the network and the actual values.

Initialize replay memory D with capacity N;
Initialize Q-function with random weights θ;
Initialize target action value function $\hat{Q}$;
for *episode = 1:M* **do**
  Initialize sequence $s_1 = \{x_1\}$ and pre-processed sequence $\phi_1 = \phi(s_1)$;
  for *t = 1:T* **do**
    With probability ϵ select a random action a_t;
    otherwise select $a_t = argmax_a(Q(\phi(s_t), a; \theta))$;
    Execute action a_t and observe reward r_t and image x_{t+1};
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and pre-process $\phi_{t+1} = \phi(s_{t+1})$;
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D;
    Sample random minibatches of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D and set

$$y_j = \begin{cases} r_j, & \text{if episode terminates at step j+1} \\ r_j + \gamma max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta_-), & \text{otherwise} \end{cases}$$

    Perform gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ w.r.t. θ;
    Every C steps reset $\hat{Q} = Q$
  end
end

Algorithm 1: Pseudo code for DQN with experience replay

4 Problem (d)

The Neural Network model has 2 layers. The first one has 16 neurons and it receives a vector with the size of the state space (4). It uses a ReLu activation function and a He uniform variance scaling initializer. The second layer uses the same initializer but a linear activation function and 2 neurons.

5 Problem (e)

Refer to code in Appendix.

6 Problem (f)

Refer to code in Appendix.

7 Problem (g)

7.1 Changing number of nodes in hidden layer

Increasing number of nodes in hidden layer causes the Q-value to be more stable. The maximum Q-value, however, does not appear to be affected by changing number of nodes. The score is concentrated in the higher regions (~ 150-200) in the case with 32 nodes, as opposed to the other 2 cases where the score fluctuates quite a bit (figures 1, 2, 3). Therefore, we used 32 nodes in hidden layer for the rest of the test cases.

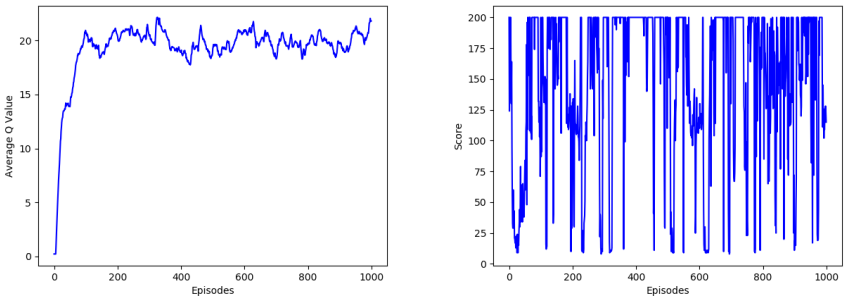

Figure 1: **Number of nodes = 8**, activation function = 'relu', lr = 0.005, discount factor = 0.95

7.2 Changing number of hidden layers

Additionally, we tried using two hidden layers instead of one with varying number of nodes.

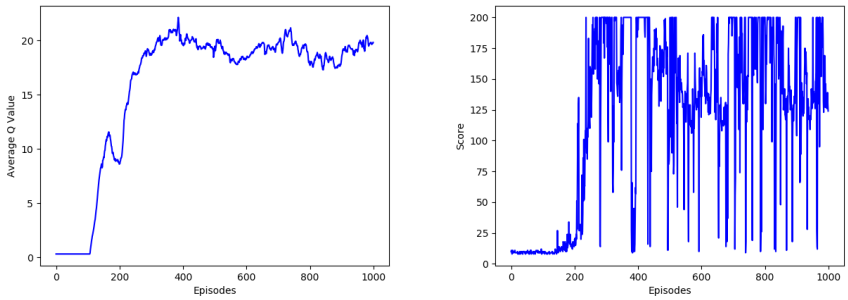

Figure 2: **Number of nodes = 16**, activation function = 'relu', lr = 0.005, discount factor = 0.95

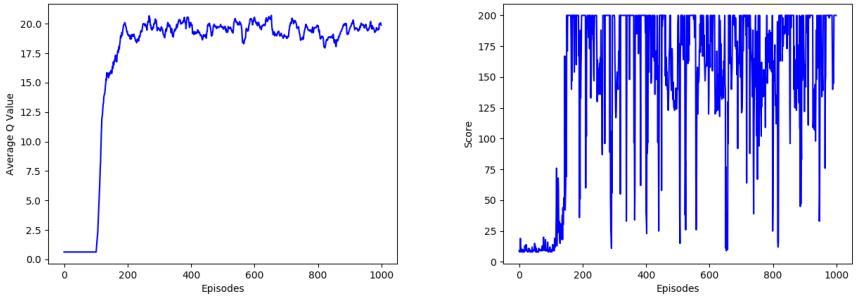

Figure 3: **Number of nodes = 32**, activation function = 'relu', lr = 0.005, discount factor = 0.95

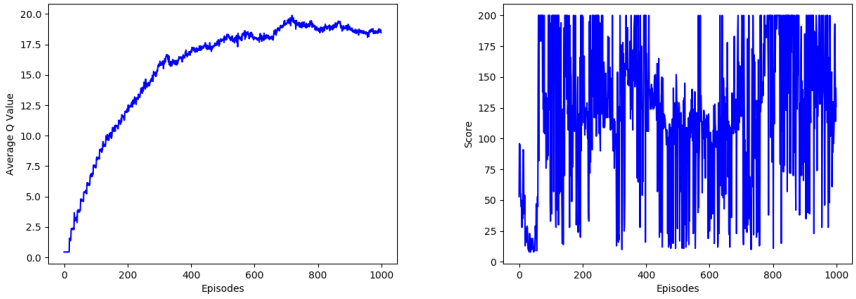

Figure 4: **Number of hidden layers = 2 (32, 16 nodes)**, activation function = 'relu', lr = 0.005, discount factor = 0.95

7.3 Changing activation function

We checked the model performance by using two activation functions - *ReLu* and *linear*. As can be seen in figures 6 and 7, relu performs better with respect to both evaluation metrics. The Q-value and score attain higher magnitude and are more

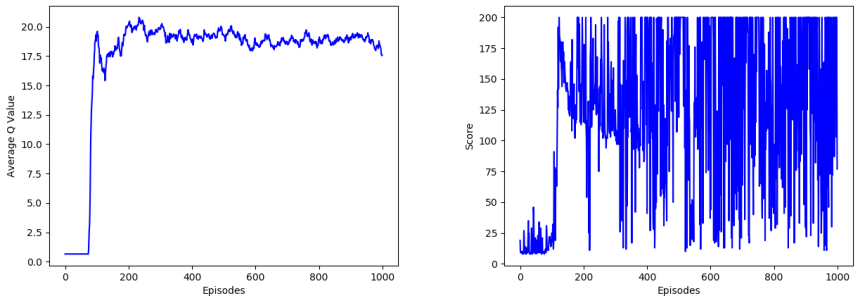

Figure 5: **Number of hidden layers = 2 (32, 32 nodes)**, activation function = 'relu', lr = 0.005, discount factor = 0.95

stable for *relu* when compared to the *linear* activation function. Using this, we decided to use *Relu* for further models.

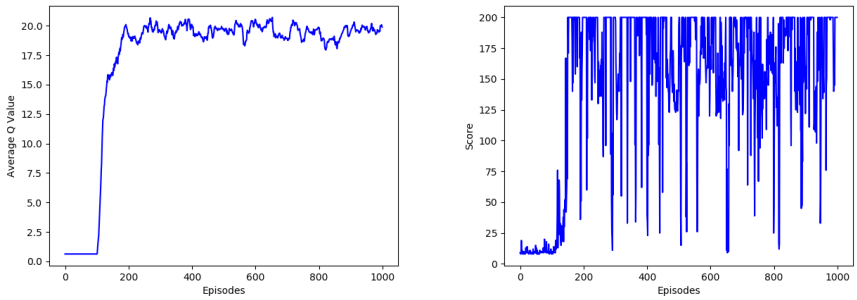

Figure 6: Number of nodes = 32, **activation function = 'relu'**, lr = 0.005, discount factor = 0.95

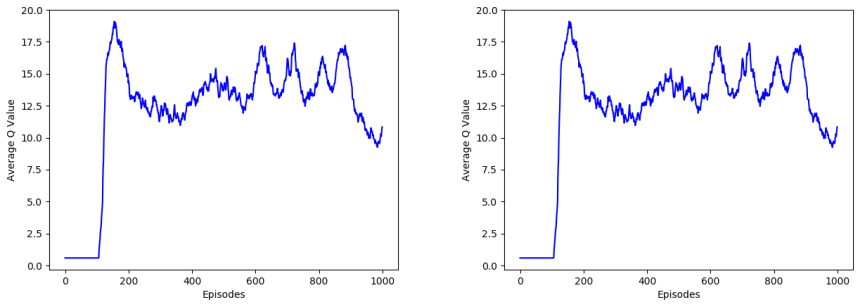

Figure 7: Number of nodes = 32, **activation function = 'linear'**, lr = 0.005, discount factor = 0.95

8 Problem (h)

8.1 Changing learning rate

As shown in figures 8, 9, 10 with a lower learning rate, the model doesn't converge within the 1000 episodes, however a higher learning rate gives less stability. While increasing the learning rate the average Q-value is higher as well as the score. However while comparing figures 10 and 6 one can notice that lr=0.001 shows a better stability on the scores and Q-values. For that reason, we will use lr=0.001 for the final model.

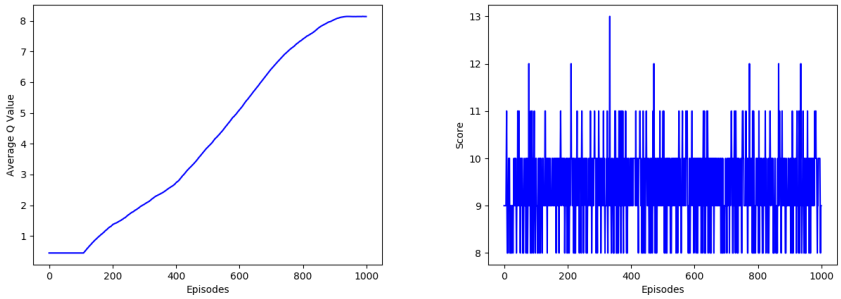

Figure 8: Number of nodes = 32, activation function = 'relu', **lr = 0.0001**, discount factor = 0.95

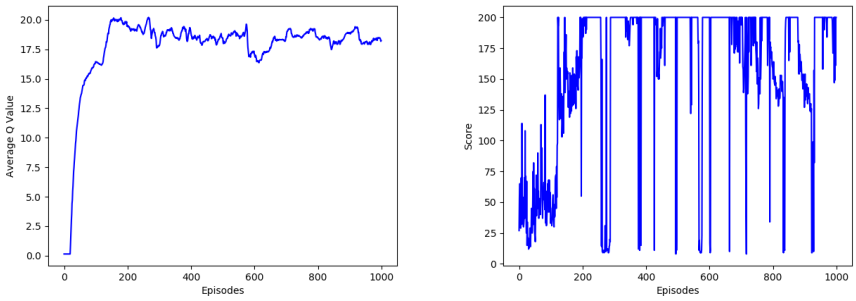

Figure 9: Number of nodes = 32, activation function = 'relu', **lr = 0.001**, discount factor = 0.95

8.2 Changing discount factor

Changing the discount factor severely affects both the Q-value and the score. A discount factor of 0.95 (figure 11) leads to the highest and most stable values for the metrics. Decreasing the discount factor from 0.95 to 0.05, causes the converged Q-value of the model to fall from $\sim$ 20 to $\sim$ 1.4, and the score to move from values

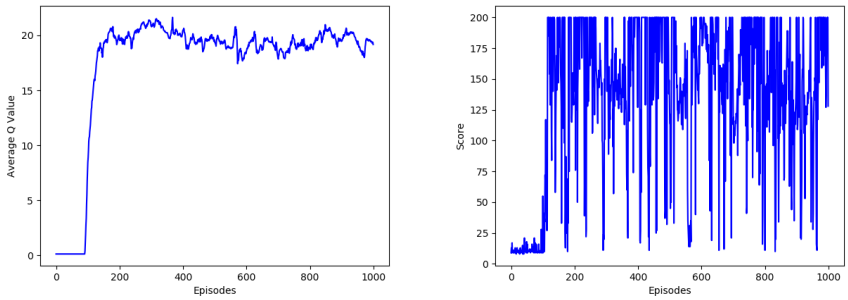

Figure 10: Number of nodes = 32, activation function = 'relu', **lr = 0.01**, discount factor = 0.95

concentrated ~ 150-200 to ~ 25-50 (figures 11,12, 13, 14 and 15). We can, therefore, conclude that higher values of discount factor work better for this probelem, and chose the value 0.95 as the optimal one.

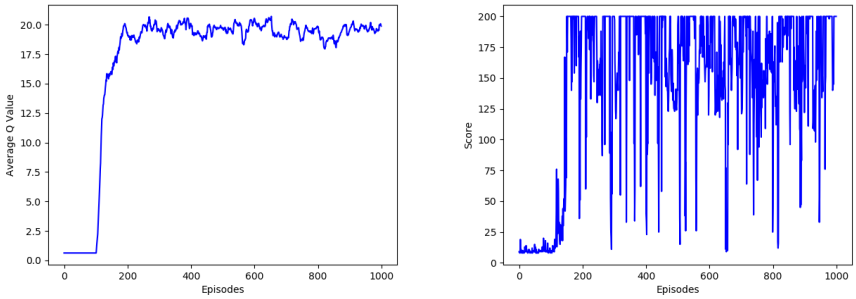

Figure 11: Number of nodes = 32, activation function = 'relu', lr = 0.005, **discount factor = 0.95**

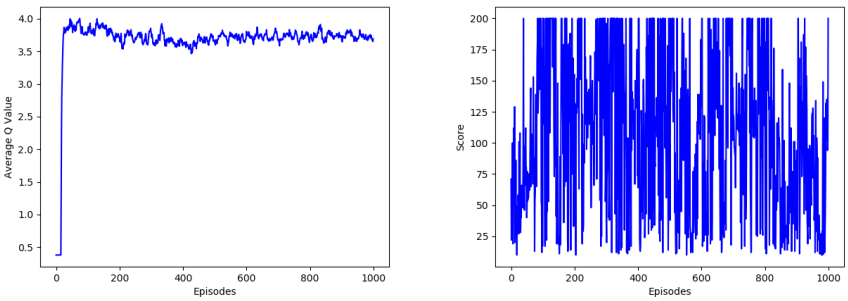

Figure 12: Number of nodes = 32, activation function = 'relu', lr = 0.005, **discount factor = 0.75**

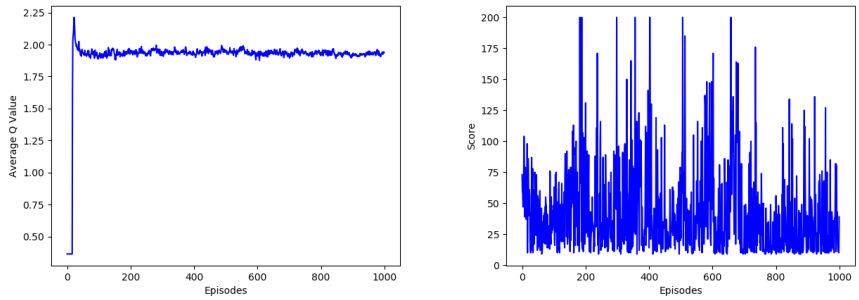

Figure 13: Number of nodes = 32, activation function = 'relu', lr = 0.005, **discount factor = 0.5**

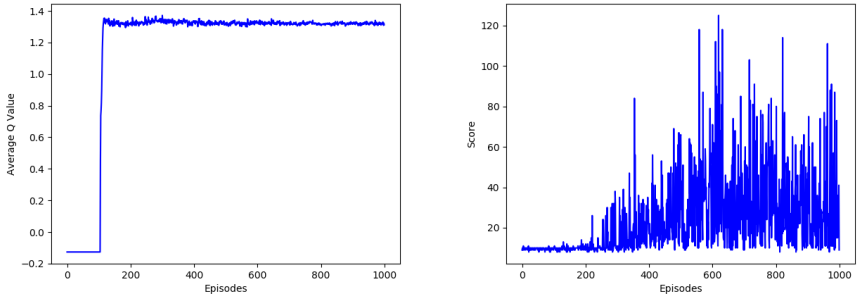

Figure 14: Number of nodes = 32, activation function = 'relu', lr = 0.005, **discount factor = 0.25**

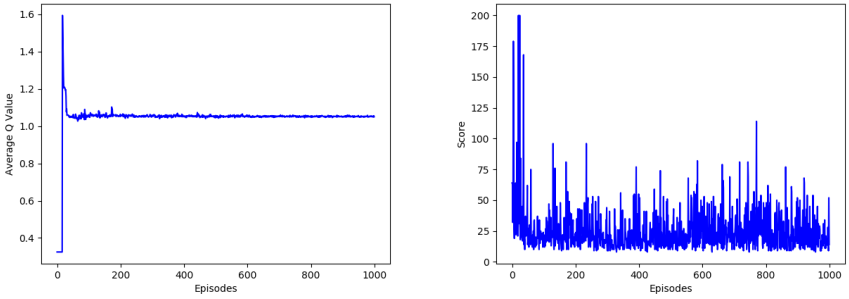

Figure 15: Number of nodes = 32, activation function = 'relu', lr = 0.005, **discount factor = 0.05**

8.3 Changing memory size

As can be seen in figures 16, 17 and 18, a small memory size (100) causes the Q-value to remain constant i.e. the model does not train at all. A memory size of 100,000 although lead to higher Q-value and score in some episode, the values were highly

fluctuating and seemed to converge around the same numbers as in the case of a memory size of 10,000. Additionally, using 100,000 lead to a significant increase in training time. Therefore, we use 10,000 for the following test.

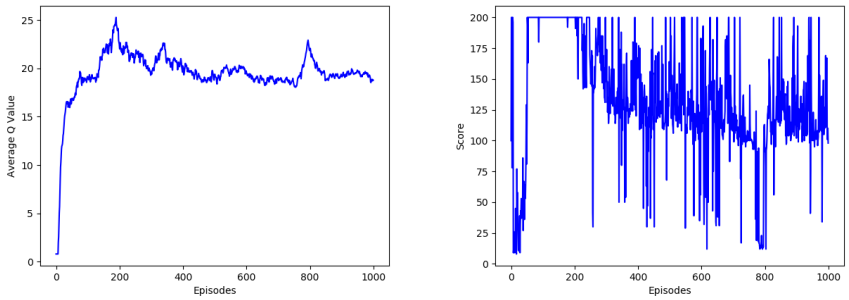

Figure 16: Number of nodes = 32, activation function = 'relu', lr = 0.005, discount factor = 0.95,**memory size = 100000**

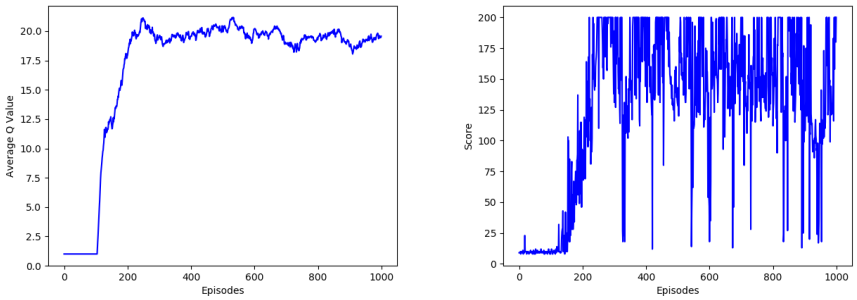

Figure 17: Number of nodes = 32, activation function = 'relu', lr = 0.005, discount factor = 0.95,**memory size = 10000**

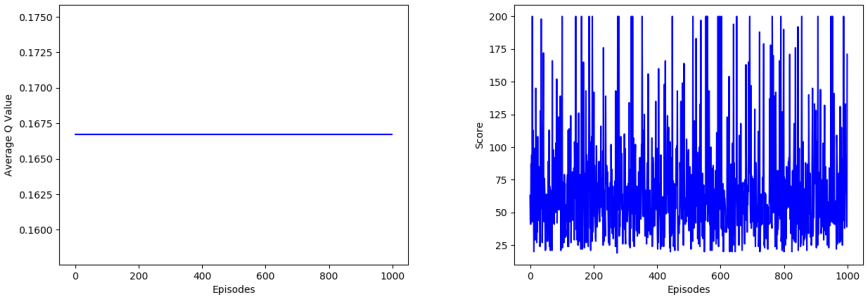

Figure 18: Number of nodes = 32, activation function = 'relu', lr = 0.005, discount factor = 0.95,**memory size = 100**

9 Problem (i)

Changing the variable *target update frequency*:
Looking at figures 3, 19 and 20, it is clear the using a frequency of 1 results in the best model performance. Increasing the frequency leads to highly fluctuating scores and overall lower Q-values.

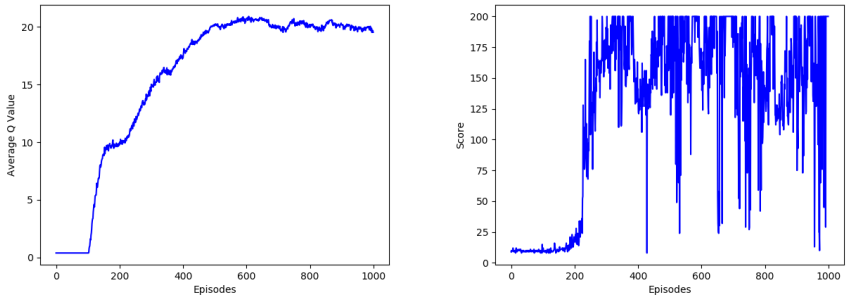

Figure 19: **frequency = 5**

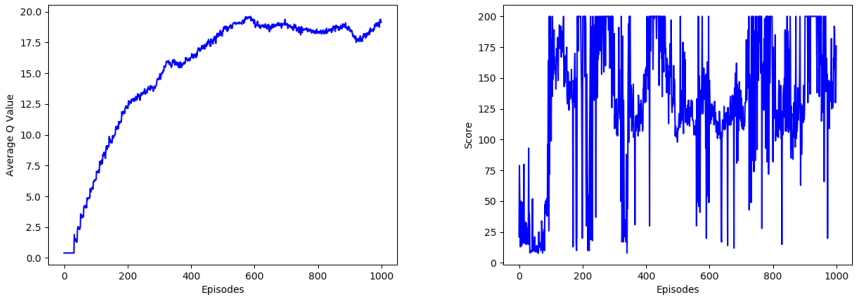

Figure 20: **frequency = 10**

10 Problem (j)

Final model parameters chosen:

- number of hidden layer = 2
- number of nodes in hidden layer = 32 (each)
- activation function - ReLu
- learning rate = 0.001

- discount factor = 0.99
- memory size = 10,000
- target update frequency = 1

The training stopped after **648** episodes. The Q-value and score plots are given below (figure 21).

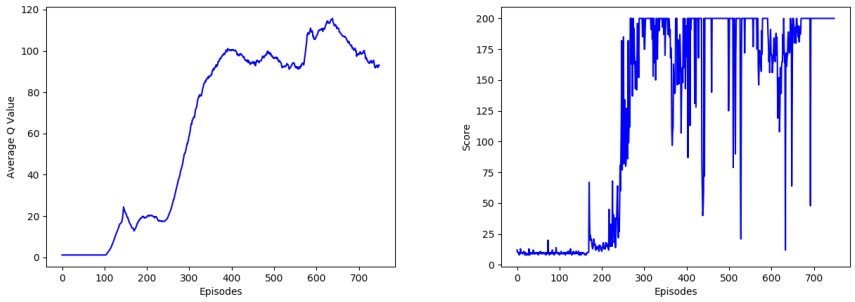

Figure 21: Number of hidden layers = 2 (32, 32 nodes), activation function = 'relu', lr = 0.001, discount factor = 0.99, memory size = 10000, frequency = 1

11 Appendix

```
import sys
import gym
import pylab
import random
import numpy as np
from collections import deque
import keras
from keras.layers import Dense
from keras.optimizers import Adam
from keras.models import Sequential

EPISODES = 1000 #Maximum number of episodes

#DQN Agent for the Cartpole
#Q function approximation with NN, experience replay, and target
    network
class DQNAgent:
    #Constructor for the agent (invoked when DQN is first called
        in main)
    def __init__(self, state_size, action_size):
        self.check_solve = True #If True, stop if you satisfy
            solution confition
```

```
        self.render = False        #If you want to see Cartpole
            learning, then change to True

        #Get size of state and action
        self.state_size = state_size
        self.action_size = action_size

       # Modify here

        #Set hyper parameters for the DQN. Do not adjust those
            labeled as Fixed.
        self.discount_factor = 0.99
        self.learning_rate = 0.001
        self.epsilon = 0.02 #Fixed
        self.batch_size = 32 #Fixed
        self.memory_size = 10000 #1000
        self.train_start = 1000 #Fixed
        self.target_update_frequency = 1

        #Number of test states for Q value plots
        self.test_state_no = 10000

        #Create memory buffer using deque
        self.memory = deque(maxlen=self.memory_size)

        #Create main network and target network (using
            build_model defined below)
        self.model = self.build_model()
        self.target_model = self.build_model()

        #Initialize target network
        self.update_target_model()

    #Approximate Q function using Neural Network
    #State is the input and the Q Values are the output.
#
    ###########################################################################

#
    ###########################################################################

        #The Neural Network model here
    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, 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):
#
    ##########################################################################

#
    ##########################################################################

        # e-greedy policy code here
        vals = self.model.predict(state)
        r = random.uniform(0, 1)
        if (r < self.epsilon or vals.all() == 0):
            action = random.randrange(self.action_size)
        else:
            action = np.argmax(vals)
        # action = random.randrange(self.action_size)
        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
#
    ################################################################################

#
    ################################################################################

        # Q-learning code here
        max_target = np.max(target_val,axis=1)
        for i in range(self.batch_size): #For every batch
            if done[i]:
                target[i][action[i]] = reward[i]
            else:
                target[i][action[i]] = reward[i] + self.
                    discount_factor*max_target[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)
        #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()
agent.plot_data(episodes,scores,max_q_mean)
```