EL282805 - Reinforcement Learning - Lab 2 - Report

Adrian Chmielewski-Anders, Leo Zeitler
9512115537, 9509289634
{amca3, llze}@kth.se

1

(a) The state space can be modeled by a vector $s = (x, \theta, \dot{x}, \dot{\theta})$ where x is the position from the center of the track and $x \in [-X, X]$ where $X \geq 2.4$m are the maximum lengths from the middle of the track. Furthermore, $\theta \in [-90, 90]$ which is measured in degrees, is the angle of the pole. $\dot{x}$ the linear velocity of the cart which is in $\mathbb{R}$. Similarly, $\dot{\theta}$ is the angular velocity at the tip of the pole which is also in $\mathbb{R}$. All state variables take values on $\mathbb{R}$ (or interval) or a discretized real line in software. Nonetheless, this makes the state space very large. The action space can be modeled as $\mathcal{A} = \{-1, 1\}$ where the agent can push from the left or right with force 1N. The reward function is defined, though unknown to the agent:

$$r_t(s,a) = \begin{cases} 1 & -12.5 \leq \pi(s,a)_\theta \leq 12.5 \text{ and } -2.4 \leq \pi(s,a)_x \leq 2.4 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

The issue is the state space is very large for tabular RL methods. Due to this, one might consider deep RL methods. The problem is modeled as an infinite time horizon. Though, the environment ends the episode after 200 steps or when or when the agent encounters a zero reward.

(b) `main` First it creates the environment, and instantiates the agent. It then initializes some test states for plotting and checking convergence. Then we go through a single training loop with N episodes, saving values to plot convergence, and going through normal training steps for each episode. That is, take an action, observe rewards and next states and update the two networks. The two plots generated are a plot of the total undiscounted reward of each episode that the agent goes through which are chosen via the ϵ-greed policy. The next graph is for each episode the maximum of the actions averaged over the same random states. In the training loop we are saving the state, action, reward, next state, and done variables into the replay buffer.

`build_model` Creates a simple neural network for approximating the Q function. The architecture is shared over θ and ϕ.

`update_target_model` Sets the target models weights ϕ to the models weights θ.

`get_action` Gets an ϵ-greedy action, initially it is completely random. However, in later parts of the assignment it is changed to be a ϵ greedy action policy with respect to Q_θ.

`append_sample` Adds a state, action, reward and next state to the replay list.

`train_model` Trains the model θ, going through each step of the Deep Q-learning. In the function `batch_size` items from the replay buffer are sampled and used as the inputs to update the network.

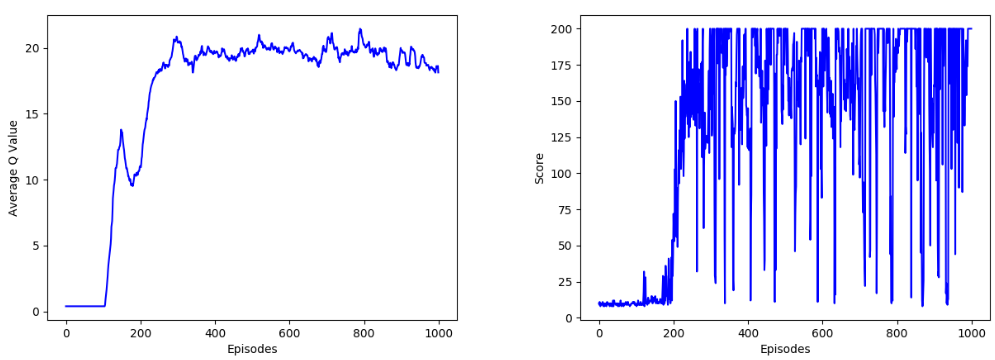
`plot_data` Is used for plotting the behavior over time of the training. Specifically for sample Q_θ values and total reward over each episode.

(c) The pseudo code in Algorithm 1 gives a outline of the deep Q-learning algorithm with experience replay. The lines 15 to 18 are equivalent to lines 192 to 199 in the given code file. Our function *Get-Action* is defined by the function `get_action`. The update of the weights of the target network is called in line 206, whereas the function is implemented

Algorithm 1 The Deep Q-Learning algorithm

function UPDATE-STEP(θ, ϕ, M, batch_size)
  for i = 1 . . . batch_size **do**
    Sample s_i, a_i, r_i, s'_i, d from M ▷ Sample (state, action, reward, next state, done) tuple from memory buffer
    if d is True **then** ▷ Check whether it was a final step
      $y \leftarrow r_i$ ▷ If so, assign the reward to the target
    else
      $y \leftarrow r_i + \lambda \max_b Q_\phi(s'_i, b)$ ▷ If not, assign current target value prediction to target variable
    $\theta \leftarrow \theta + \alpha(y - Q_\theta(s_i, a_i))\nabla_\theta Q_\theta(s_i, a_i)$ ▷ Update weights of the network according to stochastic gradient descent
function DQN(s_1, N, env, C, batch_size)
  Initialize θ, ϕ ▷ Initialize weights for target network and prediction network
  Initialize M as empty array ▷ Initialize memory buffer for experience replay
  Initialize total_steps as 0 ▷ Initialize counter that is used for updating the weights of the target network every C steps
  for e = 1 . . . N **do** ▷ Loop over all episodes
    while $t \in$ episode **do** ▷ Loop over all time steps in episode
      a_t = GET-ACTION(s_t, Q_θ) ▷ Get ϵ-greedy action
      (r_t, s_{t+1}, d) = env.STEP(s_t, a_t) ▷ Observe reward, next state and whether the task came to an end
      Append $(s_t, a_t, r_t, s_{t+1}, d)$ to M ▷ Add experience to memory buffer
      UPDATE-STEP(θ, ϕ, M, batch_size) ▷ Update weights θ of the prediction network
      total_steps $\leftarrow$ total_steps + 1 ▷ Update the step counter
      if total_steps % C == 0 **then**
        $\phi \leftarrow \theta$ ▷ Update target network weights every C iterations
  return θ

Figure 1: The two plots for the initial parameters after doing parts (e) and (f)



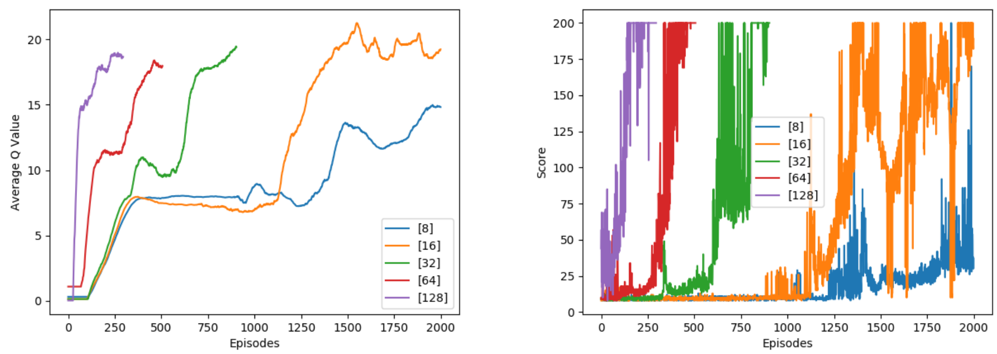
(a) The Q-plot.

(b) The total reward plot.

in `update_target_model` in line 68. The functionality of the pseudo-code function *Update-Step* is implemented in `train_model`. The calculation of the target value should be implemented between lines 114 and 124. The update of the weights θ is defined in the line 127 with the function call `self.model.fit`.

(d) The network in the `build_model` function is a two layered feed-forward neural network with 16 units in the first hidden layer and 2 in the output layer. It uses ADAM as the optimizer and uses the default activation which is linear for the output layer and the ReLU activation function for the first hidden layer. The loss function is MSE. It uses constant learning rate.

Figure 2: Experimentation with the effect of different parameters on the network complexity.



(a) The Q-value plot for a shallow network with a single hidden layer and different number of neurons

(b) The score plot for a shallow network with a single hidden layer and different number of neurons

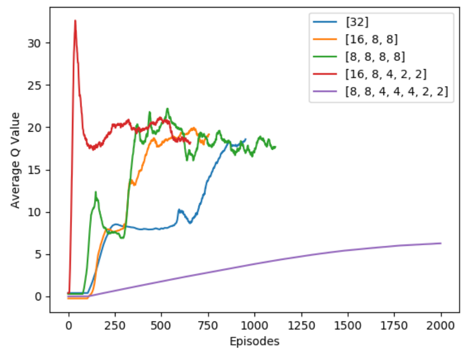
(e) See the attached code.

(f) The code for parts (e) and (f) is appended below. The two plots with all the default parameters are shown in Figure 1

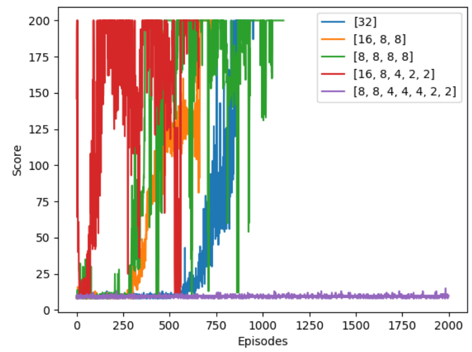
(g) We investigated the impact of the number of hidden neurons and the depth of the network on the performance of the system. As mentioned in the task description, we did some parameter fiddling in order to make the agent solve the task. We discovered that a buffer size of 5000, 2000 epochs and a learning rate set to 0.0005 shows some decent results. We used these parameter settings in all following experiments conducted within the task (g). Firstly, we examined the effect of the number of hidden neurons organised in shallow network with a single hidden layer on the learning. We stopped the learning procedure as soon as the task was considered to be solved, namely when the average reward over 100 consecutive episodes is greater than or equal to 195 to have a measure for the time to converge. As it is depicted in figure 2 the algorithm converges quicker the more hidden neurons are involved. Although the value that the Q-function approaches does not differ for different architectures, it can be seen in the figure 2b that the network with 128 hidden neurons reaches quickly to a score of 200, whereas the network with only 8 is not able to solve the task within 2000 learning epochs. This suggests that a higher number of hidden neurons makes it easier to learn the parameters need to find an accurate mapping from the state space to a corresponding value. Yet it should be noted that the time it takes for the network to learn parameters that are able to solve the task is heavily dependent on the random samples that are drawn from the memory buffer. Thus, the plots are slightly different in different trials, and it might happen in some runs the network with 8 neurons does converge in a lower number of iterations. Nevertheless, even within the trial-to-trial variance we could discover that the number of hidden neurons is (to a certain extent) proportional to the time it takes the network to solve the task.

Leaving the number of hidden units constant while increasing the depth of the network does not give such clear results. In a second experimental setup we kept the number of hidden neurons constant at 32, but increased the depth of the network to the following architectures: [32], [16, 8, 8], [8, 8, 8, 8], [16, 8, 4, 2, 2], and [8, 8, 4, 4, 4, 2, 2], where the number in the lists represents the number of units in the particular hidden layer. The Q-values of the initial state as well as the scores are shown in the graphs in figure 3. The results obtained from the experiments are a mixed bag and do not allow a general statement. While the deep network with 5 hidden layers performed better than the shallow network, it converged quicker than the networks with 4 and 7 hidden layers. It should be stressed that these particular results are random and that we face a trial-to-trial variance. Nonetheless, we were not able to deduce any particular trend form the experiments. Hence, we will use a shallow network in the following tasks.

Figure 3: Experimentation with the effect of different parameters on the network complexity.

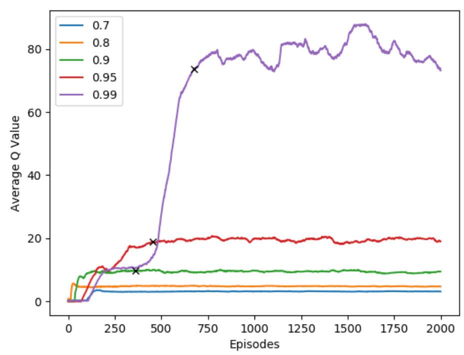


(a) The Q-value plot for different deep architectures.

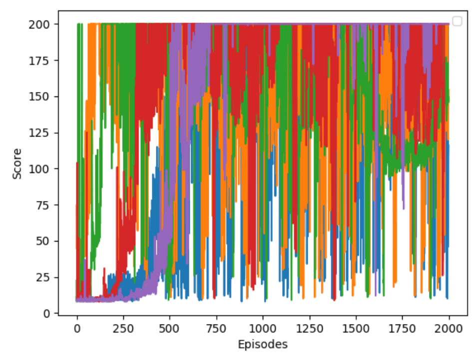


(b) The score plot for different deep architectures.

Figure 4: Effects of discount factor on training

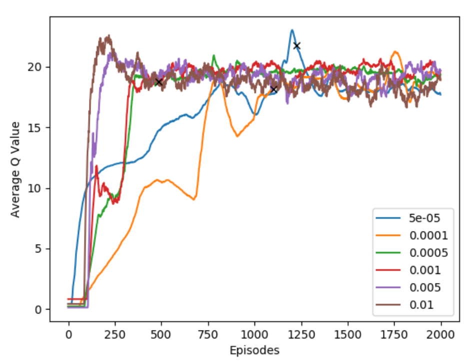


(a) The Q-value plot

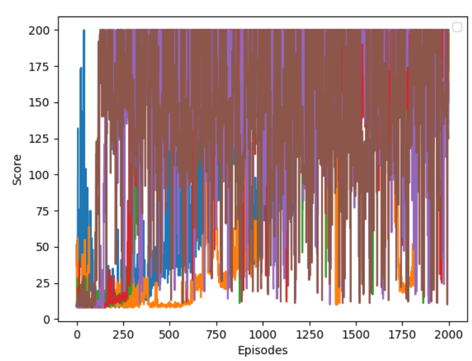


(b) The score plot

Figure 5: Effects of learning rate on training

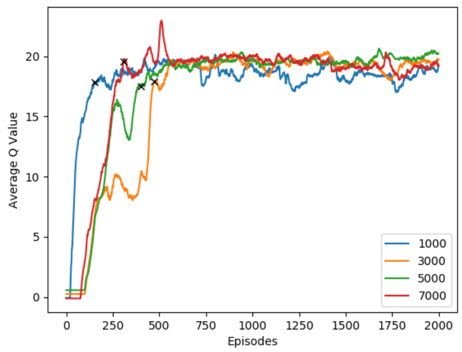


(a) The Q-value plot

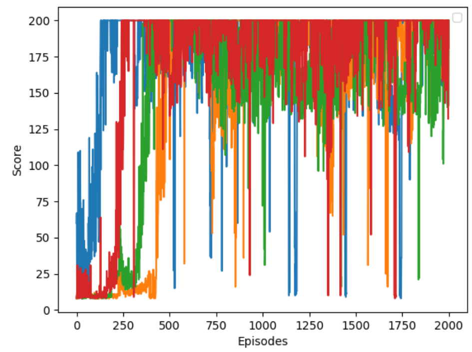


(b) The score plot

Figure 6: Effects of memory size on training



(a) The Q-value plot



(b) The score plot

(h) Figures 4, 5, and 6 show experiments examining the effects on training when changing the discount factor, learning rate, and memory size, respectively. For all three experiments, a variety of different parameter values were tried while keeping all other variables constant. The model that was selected had one hidden layer with 128 units, discount factor of 0.95, learning rate of 0.0005, memory size of 5000 and update frequency of 1. Put in other words, each experiment changed one of these variables but left all others unchanged. Each training run lasted 2000 episodes. The check solve flag was turned on, and a black "X" on each of the graphs marks the point where this configuration had solved the problem.

Discount factor: For lower values of λ the problem never was solved, in our experiments these values were $\lambda = 0.7, 0.8$. The results show that for discount factors which did result in a solved game, the higher the discount factor, the more episodes it took to solve. It is possible that this is because higher discount factor leads to policies which have more weight on future reward. Larger discount factors had better scores on average than smaller discount factors. Also the Q-values for larger values of λ are larger, which intuitively makes sense because the Q-function grows with increasing λ.

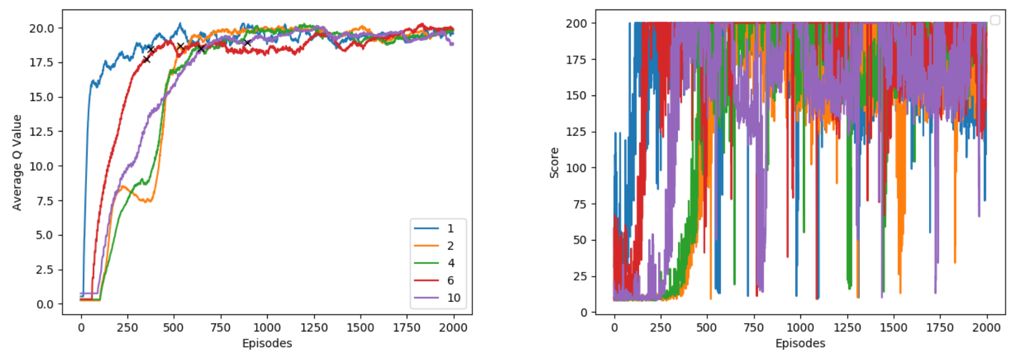
Learning rate: As one would expect, with higher learning rate, there is much more noise in the average Q values since θ is changed more each update. Lower learning rates solved the task later. That is, after experiencing more episodes. On the other hand, high learning rates did not solve the problem in some cases at all.

Memory size: The results can be seen in Figure 6. Generally all values used for memory size solved the problem, in fact after around the same number of episodes. The main difference was the swings in the average Q-values where it was observed that smaller memory size had larger swings in average Q-value. One reason for this might be that for larger memory size, the sampling is more "uniform" and thus in the long term average won't cause the Q-values to fluctuate so much, whereas smaller memory size have more correlated randomness since they are fewer values to sample from and they all happened more recently. This is the case because the memory buffer is like a sliding window. For all the experiments a value of 5000 was chosen because it was in the middle of values that did work, and it worked consistently, and was not as noisy as smaller memory sizes.

(i) Using the default values for the model as specified in part (h) the update frequency was changed and results/effects on training can be seen in Figure 7. Interestingly, all solve the problem, just after different number of observed episodes. The best seemed to be a frequency of 6. Generally, the lower the value (i.e. higher frequency) the faster change and more noise in the average Q-values, and scores. Oddly, this is not the case for the highest value we tried of 10.

(j) The parameters chosen for the previous parts do solve the problem, as do many others as made evident by the plots shown. For example, the values which we used as baselines for parts (h) and (i) solve the problem.

Figure 7: Effects of update frequency on training



(a) The Q-value plot

(b) The score plot

2 Code

The code after completing parts (e) and (f) is appended below.

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

EPISODES = 2000 #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 = 0.95
        self.learning_rate = 0.0005
        self.epsilon = 0.02 #Fixed
```

```python
        self.batch_size = 32 #Fixed
        self.memory_size = 5000
        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.
###############################################################################
###############################################################################
        #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(128, 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)
        if random.random() < self.epsilon:
            return action
        else:
            Q_actions = self.model.predict(state)
```

```python
            return np.argmax(Q_actions, axis=1)[0]


    ###############################################################################
###############################################################################
    #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
            if done[i]:
                target[i][action[i]] = reward[i]
            else:
```

```
                target[i][action[i]] = self.discount_factor * np.max(target_val[i]) + reward[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)
```

The same code with additional functions to graph and with our experiments follows:

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

EPISODES = 2000 #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, target_update_frequency=1, arch=[16], discount_factor=0.95, learning_rate=0.0005, mem_size=5000):
        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 = discount_factor
        self.learning_rate = learning_rate
        self.epsilon = 0.02 #Fixed
        self.batch_size = 32 #Fixed
        self.memory_size = mem_size
        self.train_start = 1000 #Fixed
        self.target_update_frequency = target_update_frequency

        #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)

        self.arch = arch
        #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.
################################################################################
################################################################################
        #Edit the Neural Network model here
        #Tip: Consult https://keras.io/getting-started/sequential-model-guide/
    def build_model(self):
        model = Sequential()
        for num_units in self.arch:
            model.add(Dense(num_units, 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)
        if random.random() < self.epsilon:
            return action
        else:
            Q_actions = self.model.predict(state)
            return np.argmax(Q_actions, axis=1)[0]

        ################################################################################
################################################################################
    #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
```

```python
        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
            if done[i]:
                target[i][action[i]] = reward[i]
            else:
                target[i][action[i]] = self.discount_factor * np.max(target_val[i]) + reward[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, dir_name, arch=None):

        if not os.path.exists(dir_name):
```

```
            os.makedirs(dir_name)

        if arch is None:
            subname = ''
        else:
            subname = str(arch)
        pylab.figure(0)
        pylab.plot(episodes, max_q_mean, 'b')
        pylab.xlabel("Episodes")
        pylab.ylabel("Average Q Value")
        pylab.savefig("%s/qvalues-%s.png" % (dir_name, subname))

        pylab.figure(1)
        pylab.plot(episodes, scores, 'b')
        pylab.xlabel("Episodes")
        pylab.ylabel("Score")
        pylab.savefig("%s/scores-%s.png" % (dir_name,  subname))

def plot_data_multiple(episodes, scores, max_q_mean, solved_times,
    dir_name, names):

    if not os.path.exists(dir_name):
        os.makedirs(dir_name)

    pylab.figure(0)
    pylab.clf()
    for i in range(len(episodes)):
        pylab.plot(episodes[i], max_q_mean[i], label=names[i])
    actually_solved_times = [time for time in solved_times if time
        != -1]
    solved_values = [max_q_mean[i][solved_times[i]] for i in
        range(len(solved_times)) if solved_times[i] != -1]
    pylab.plot(actually_solved_times, solved_values, 'kx',
        label='solved')
    pylab.xlabel("Episodes")
    pylab.ylabel("Average Q Value")
    pylab.legend(names)
    pylab.savefig("%s/qvalues.png" % dir_name)

    pylab.figure(1)
    pylab.clf()
    for i in range(len(episodes)):
        pylab.plot(episodes[i], scores[i])
    pylab.xlabel("Episodes")
    pylab.ylabel("Score")
    pylab.legend()
    pylab.savefig("%s/scores.png" % dir_name)

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

def simulate(agent, times):
    env = gym.make('CartPole-v0')
    env = wrappers.Monitor(env, directory='sims/trained',
        force=True)

    for run in range(times):
        state = env.reset()
        state_size = env.observation_space.shape[0]
```

```python
        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]]

        done = False
        while not done:
            #env.render()  # Show cartpole animation

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


def train(arch, discount_factor=0.95, learning_rate=0.0005, mem_size=5000, update_freq=1):
    solved = -1
    #For CartPole-v0, maximum episode length is 200
    env = gym.make('CartPole-v0') #Generate Cartpole-v0 environment object from the gym library
    #env = wrappers.Monitor(env, directory='sims/training', force=True)

    #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, arch=arch, discount_factor=discount_factor, learning_rate=learning_rate, mem_size=mem_size, target_update_frequency=update_freq)

    #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 episodesepisode, 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)

                if e % 100 == 0:
                    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 and solved == -1:
                    if np.mean(scores[-min(100, len(scores)):]) >= 195:
                        print("solved after", e-100, "episodes")
                        solved = e-100
                        # agent.plot_data(episodes,scores,max_q_mean[:e+1],'part-g
                        env.close()
                        simulate(agent, 3)
                        return episodes, scores, max_q_mean, solved
```

```
    env.close()
    #agent.plot_data(episodes,scores,max_q_mean, '')
    return episodes, scores, max_q_mean, solved


if __name__ == '__main__':
    episodes = []
    scores = []
    max_q_means = []
    # num_unit_values = [16, 32, 64]
    # archs = [[8, 32], [16, 32, 32], [16, 32, 32, 32]]
    # archs = [[8], [16], [32], [64], [128]]
    # archs = [[8]]
    #names = list(map(str, archs))

    # discount factor tests
    # discount_factors = [0.7, 0.8, 0.9, 0.95, 0.99]
    # names = list(map(str, discount_factors))
    arch = [128]
    # learning_rates = [0.00005, 0.0001, 0.0005, 0.001, 0.005, 0.01]
    # names = list(map(str, learning_rates))
    # mem_sizes = [500, 1000, 5000, 9000]
    # names = list(map(str, mem_sizes))

    # update_freqs = [1, 2, 4, 6, 10]
    # names = list(map(str, update_freqs))
    #
    # solved_times = []
    #
    # for update_freq in update_freqs:
    #     eps, score, qs, solved = train(arch, discount_factor=0.95, learning_rate=0.0005, mem_size=5000, update_freq=update_freq)
    #     episodes.append(eps)
    #     scores.append(score)
    #     max_q_means.append(qs[:len(score)])
    #     solved_times.append(solved)
    # plot_data_multiple(episodes, scores, max_q_means, solved_times, 'update-freq', names)

    train([128], discount_factor=0.95, learning_rate=0.0005, mem_size=5000, update_freq=1)
```