EL2805 Reinforcement Learning

Alice Karnsund
Therese Stålhandske

Abstract

This is the Lab Assignment 2 for the course Reinforcement Learning EL2805.

1 Formulation of the RL problem

Our goal in this lab is to train an agent that finds an optimal policy that prevents the pole on the cart from falling over, see Figure 1. To begin with, we formulate the RL problem, i.e. the state-space, action-space etc.

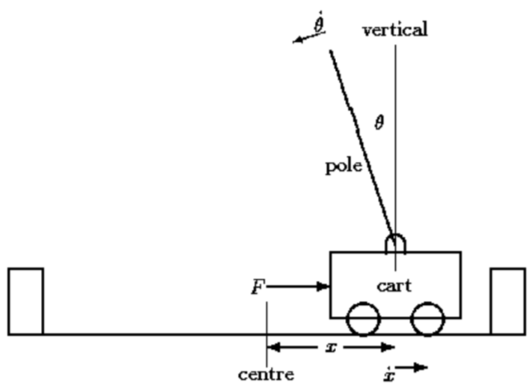

Figure 1: The cart and pole system.

Each state is described by four observations made at each time step, the cart position, cart velocity, the pole angle and the pole velocity at the tip of the pole. In Table 1, these observations are described along with their min and max values.

Observation	Min	Max
Cart position, x	-4.8	4.8
Cart velocity, $\dot{x}$	$-\inf$	$\inf$
Pole Angle, θ	$-24°$	$24°$
Pole velocity, $\dot{\theta}$	$-\inf$	$\inf$

Table 1: States

The action space constitutes of two actions, either push the cart to the left or to the right, with a magnitude of 1N, to balance the pole.

$$A = \{L = \textit{push to the left}, R = \textit{push to the right}\}$$

The agent can receive two different rewards, which depends on the pole angle θ and the cart position x. These are represented in Table 2.

State s_t	Reward r_t
$\theta < 12.5°$ and $x < 2, 4m$	1
Otherwise	0

Table 2: Rewards

Furthermore, we are considering an episodic task. An episode terminates after 200 time steps or once the agent receives 0 rewards for the first time.

For this problem, we cannot use standard methods to solve the problem efficiently. This is due to the fact that we are dealing with an infinite state space, as can be seen from Table 1.

2 Code description

The file `cartpole_dqn.py` contains one class, `DQNAgent`, that has seven functions in it, and a `__main__` method.

__main__

First, we create the cart-pole environment with the use of the `gym`-library. Thereafter we get the state and action sizes defined by the environment. We create a new agent, an instance of the class `DQNAgent`. The first for-loop collects a set of test states for plotting the Q values using uniform random policy. For the first iteration, `done = True`, and we will enter the if-statement and there set the state to the initial one. Thereafter we will enter the else-statement, until `done` is change, where we will select a random action and then observe the new state reward, done and info, after taking this action. The `done` variable will be set to `True` again if we reach the end of an episode or if the pole has tipped too far for example.

The second for-loop goes over all the episodes, collecting cumulative scores for each episode. For every episode, we start in the initial state. As long as the episode is not over, in each time step we select an action, line `3` in Algorithm 1, and collect a sample (s_t, a_t, r_t, s'_t) that is added to the replay buffer, lines `4-5` in Algorithm 1. Thereafter we train the agent, using `train_model` and add the reward to this episodes total reward, lines `6-12` in Algorithm 1. If the episode is over, `done` is set to `True` and we update the target network and collect this episode's cumulative reward to the score list, line `14` in Algorithm 1. Finally, if we have reached a satisfying solution we plot the solution using `plot_data` and then terminate the algorithm. This corresponds to lines `15-17` in Algorithm 1. If we do not reach a satisfying solution we will simply plot what we have.

Behavior of each function

The class `DQNAgent` has seven functions in it.

`__init__(self, state_size, action_size)`
First of all the constructor takes as arguments the state size, i.e. the number of elements representing a state, and the number of actions, in this case, 2. The class holds 16 instance variables. The first two are boolean variables, where the first one is `True` if we have reached the solution condition. The other one simply allows us to see the simulation of the learning. The next two variables hold the state and action size, as given by the arguments. Thereafter we have seven instance variables that set the hyperparameters for the DQN. `discount_factor, learning_rate, epsilon, batch_size, memory_size, train_start` and `target_update`. Here `memory_size` is the size of the replay buffer and `memory_start` simply assures that we do not start training if we do not have enough memory. The last five, `test_state_no, memory, model` and `target_model` together with `update_target_model()`, these sets the the number of test states for the Q-value plots, creates a replay buffer and creates two different ANNs for the two Q-functions, Q_ϕ and $Q_{\phi'}$.

`build_model(self)`
The function `build_model` returns an Artificial Neural Network with one hidden and one output

layer, which makes use of a ReLu activation function and a linear activation function, respectively. The ANN is compiled using the mean square error and the stochastic optimization method Adam.

`update_target_model(self)`
This function updates the weights for the target model, $Q_{\phi'}$, to be the same as those for the model Q_{ϕ}. Q_{ϕ} is the function approximation of the (state, action) value function of a given policy π, Q^{π}.

`get_action(self,state)`
In this function we provide an ϵ-greedy policy that will, given a state, return an action. This corresponds to line `3` in Algorithm 1.

`append_sample(self, state, action, reward, next_state, done)`
This function takes care of adding samples to the replay buffer. Where each sample holds the observations, (s_t, a_t, r_t, s'_t). The replay buffer is of fixed size, therefore old experience will be removed as new is observed. This corresponds to line `5` in Algorithm 1.

`train_model(self)`
This function is responsible for the training part and is also where we will provide our Q-learning code. First the function makes sure that we have enough memory to start training, i.e. we must have a full replay buffer. We decide upon a `batch_size` that is at most equal to the size of the replay buffer. Then we sample `batch_size` samples from our replay buffer into `mini_batch`, which will be used for training. Now `mini_batch` contains samples of the form (s_i, a_i, r_i, s'_i). We divide this information into two arrays, `update_input` that holds all s_i and `update_target` that holds all s'_i. And two lists, `action` that holds all a_i and `reward` that holds all r_i. Then we predict an output for the Q_{ϕ} network, using `update_input` as input, and another output for the $Q_{\phi'}$ network, using `update_target` as input. Then after providing our Q-learning code with the target values, we will train the Q_{ϕ}. This corresponds to lines `6-11` in Algorithm 1.

`plot_data(self, episodes, scores, max_q_mean)`
This function creates two plots, one showing the average Q-value over episodes and the other one showing the score over episodes.

3 Neural Network description

Keras is a neural network API that is able to run together with TensorFlow, CNTK, or Theano. It offers easy and fast modeling with Artificial Neural Networks.

For this task, we are using a sequential model to build our network. This means that we are using a linear stack of layers. To stack layers one simply uses the `.add()` function. In our case, we are using two layers, one hidden layer, and one output layer. Note that the counting index starts with the first hidden layer up to the output layer, thus the "input layer" is not counted.

The hidden layer is a `Dense` layer, i.e. a fully connected layer, and takes as input a vector of dimension 4, i.e. the `state_size`. The number 16 is the dimensionality of the output space and means that the layer has 16 hidden units. Furthermore, the hidden layer makes use of an ReLu activation function, $f(x) = max(0, x)$ when computing the output of each neuron. Lastly, the initial weight values are set according to `"he_uniform"`. That is, weights are drawn from a uniform distribution within $[-limit, limit]$ where the limit is $\sqrt{(6/fan\_in)}$, where fan_in is the number of input units in the weight tensor [1].

The output layer is also a fully connected layer, which takes as inputs the outputs of the hidden layer, and therefore has an input dimension of 16. The output layer consists of 2 units, i.e. `action_size`. Thus the output of the network is an array holding the Q-values for each action of the state that was given as input to the network. This layer makes use of a linear activation function when calculating the output of each neuron, $f(x) = x$. And the initial weights are set in the same way as those for the hidden layer.

The whole model makes use of the stochastic optimization method Adam and aims to minimize the Mean Least Square error, both given as an argument to `.compiler()`. Where the `.compiler()` configures the model for training.

[1] https://keras.io/initializers/

4 Modification of code

`get_action(self, state)`
DQN is an off-policy algorithm. This means that we evaluate or improve a policy different from that used to generate the data. Here we generate data w.r.t Q_ϕ, but the target policy is w.r.t $Q_{\phi'}$. Due to this ϵ can be fixed and do not have to decrease over time, in this case, $\epsilon = 0.02$, always. We make use of an ϵ-greedy behavior policy,

$$a = \left\{ \begin{array}{cc} \text{argmax}_b Q_\phi^{(t)}(s, b), & \text{w.p } 1-\epsilon \\ \text{uniform}(A_s), & \text{w.p } \epsilon \end{array} \right\} \tag{1}$$

`train_model(self)`
In this part we provided the targets, y_i for the Q_ϕ network.

$$y_i = \left\{ \begin{array}{lr} r_i & \texttt{if episode stops in } s_i' \\ r_i + \lambda max_a Q_{\phi'}(s_i', a) & \texttt{otherwise} \end{array} \right\} \tag{2}$$

These are then inserted into the `target` vector, where y_i is set as target for $Q_\phi(s_i, a_i)$. Here a_i is the action taken in s_i.

5 Assessment of model

N hidden layers	1
N neurons	16
Discount factor	0.95
Learning rate	0.005
Memory size	1000

Table 3: Default Hyper-parameter settings

The default hyper-parameters, shown in Table 3, are used for the learning procedure demonstrated in Figure 2. It seems like the agent are able to learn how to keep the pole up, but it is very inconsistent having large score fluctuations over the run episodes. Clearly, these set of hyper-parameters does not solve the problem. It was now our task to extrapolate these hyper-parameters in order to improve the learning performance.

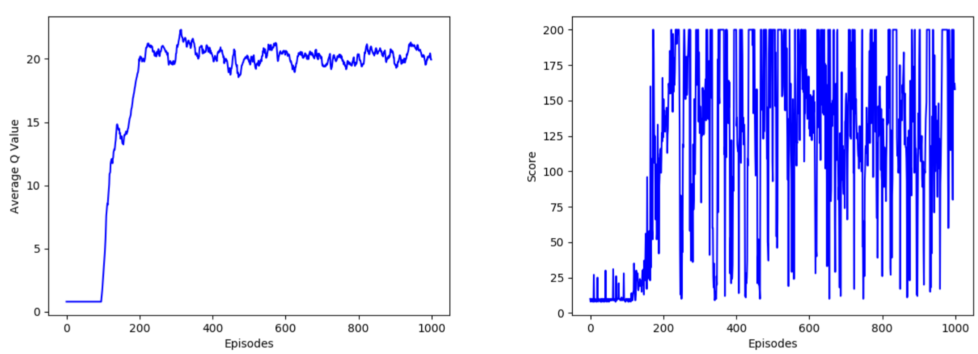

Figure 2: Plot of the average Q-value over episodes (left) and plot of the score over episodes (right), using the default hyper-parameters.

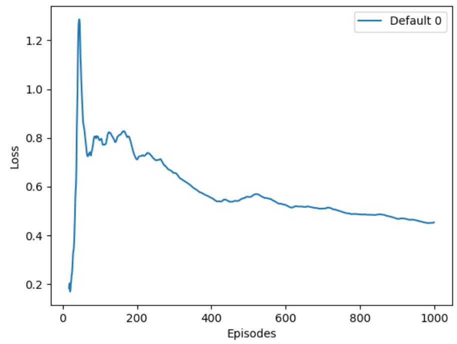

Figure 3: Plot over the average loss function using the default hyper-parameters.

First, we looked at increasing the number of hidden layers to 2 and 3, with different number of hidden neurons in each layer. We also tried different sizes of the replay buffer together with this. We noticed that an overall general solution was best found using one layer with an increased amount of neurons. We found that 84 neurons gave good result after a good amount of trial and error, thus we decided to use this and base further investigations on this setting. The number of neurons is strongly dependent on the learning rate and the discount factor of the learning rate.

6 Investigation

We investigate the three hyper-parameters learning rate, discount factor and memory size and their respective effect on the learning performance. This is done with three representative simulations, varying one parameter at the time, while keeping the other two fixed. Further, we also investigate how the target update frequency effect the training. All the following investigations are based on a model with one hidden layer and 84 neurons.

6.1 Learning Rate

The learning rate is a measure of how far to move the weights in the direction of the gradient for a mini batch. Our intuition then tells us that a low learning rate will give a more reliable, but also slower, learning process. If the learning rate is too high, the training might not even converge as it can overshoot the minimum as the steps are to large.

The loss functions for three different learning rates are displayed below in figure 5. In this graph, the differences between a low and a high learning rate are clearly shown. When the learning rate $= 0.0005$, we see a bad performance in early episodes, where the loss function dramatically increases. But as the number of episodes increases, the loss function starts to steadily decrease. For learning rate $= 0.05$, we see the opposite behavior, the loss function starts low, but when about 400 episodes have elapsed, the loss functions start increasing, indicating a poor learning performance. Learning rate 0.005, have a behavior somewhere in between and it shows a steady loss function over the first 1000 episodes.

In Figure 4, we see the score and the average q values for each of the learning rates. Considering the score plot, we see that the graph representing learning rate $= 0.0005$, shows the most stable behavior, even if it takes a little more than 400 episodes getting to a score of 200, which in comparisons is quite slow.

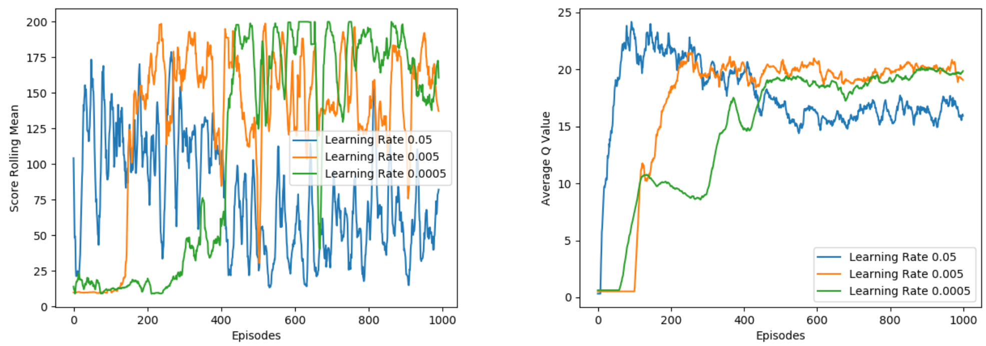

Figure 4: (Left) Rolling mean of the scores per episode for different learning rates. (Right) The Average Q-value per episode for different learning rates.

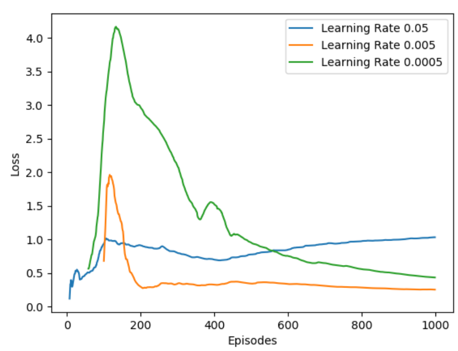

Figure 5: Mean loss function for different learning rates.

6.2 Discount Factor

The discount factor regulates the length of the time horizon in which that the agent should consider the rewards. A low discount factor premiers immediate rewards, while a higher discount factor also considers rewards further into the future.

Different discount factors are demonstrated in the figures below. With a discount factor closer to 1, 0.99, we can see that the average Q-value, in Figure 6 to the right, strongly increases and do not seem to converge at the same pace as the lower discount factors 0.90 and 0.95. This also leads to larger fluctuation in the score plot, shown in Figure 6 to the left. Looking at loss function, in figure 7, the loss for a discount rate of 0.99 increases together with the number of episodes. This is clearly not a desirable behavior.

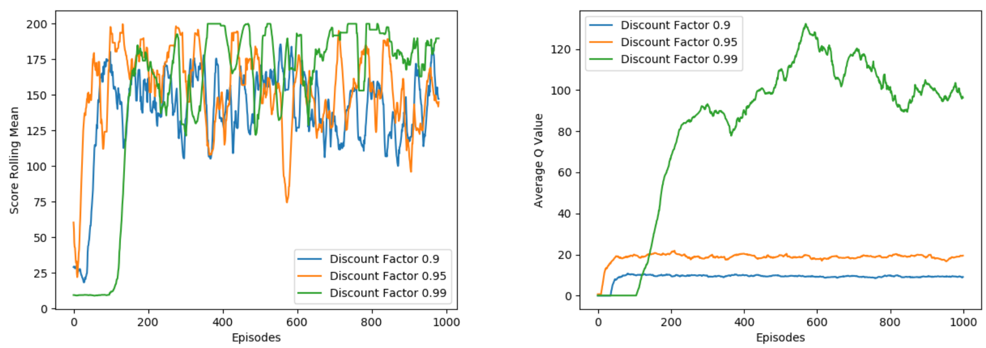

Figure 6: (Left) Rolling mean of the scores per episode for different discount factors. (Right) The Average Q-value per episode for different discount factors.

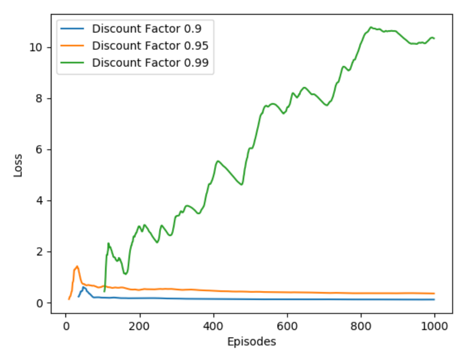

Figure 7: Mean loss function for different discount factors.

6.3 Memory Size

The size of the memory determines the number of experiences that the agent will base its next action on. The size of the optimal memory depends, among other things, on the variability of the training samples. For simulation purposes, we try to increase the memory with an exponential of 10. The most distinguishable difference between the different memory sizes, is that when the memory size is smaller. Then the learning scores, shown in 8, have higher fluctuation. The scores have a higher tendency to drop, even in later episodes. This might indicate that, in this case, a larger memory size is preferred.

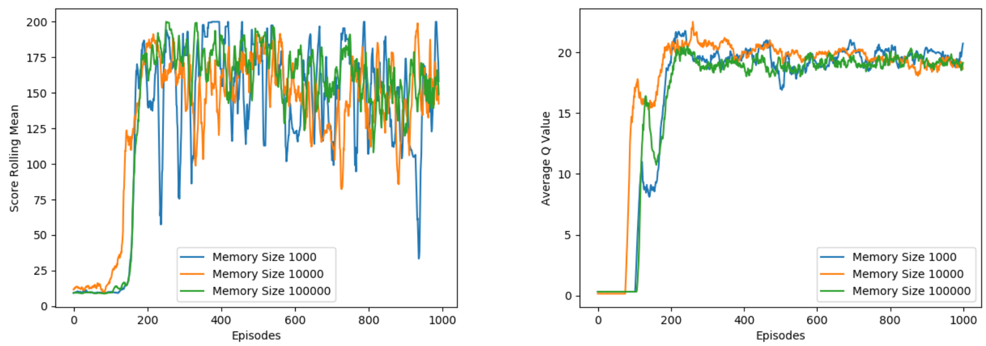

Figure 8: (Left) Rolling mean of the scores per episode for different memory sizes. (Right) The Average Q-value per episode for different memory sizes.

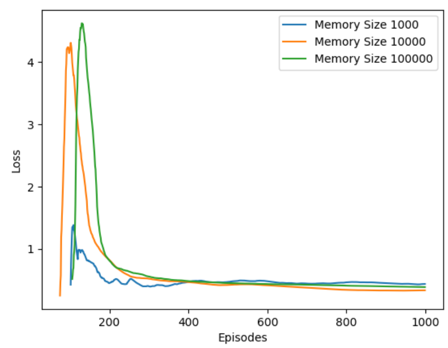

Figure 9: Average loss function for each episode for different memory sizes.

6.4 Target Update Frequency

The target update frequency controls how often the target network is updated. In Figure 10, to the left, we see the scores over the episodes. Naturally, when the target update frequency $= 100$, it will take a larger number of episodes to learn the true behavior, as the updates are not as frequent. An interesting thing though, can be seen in figure 11, where the loss function decreases rather fast for a frequency of $= 100$.

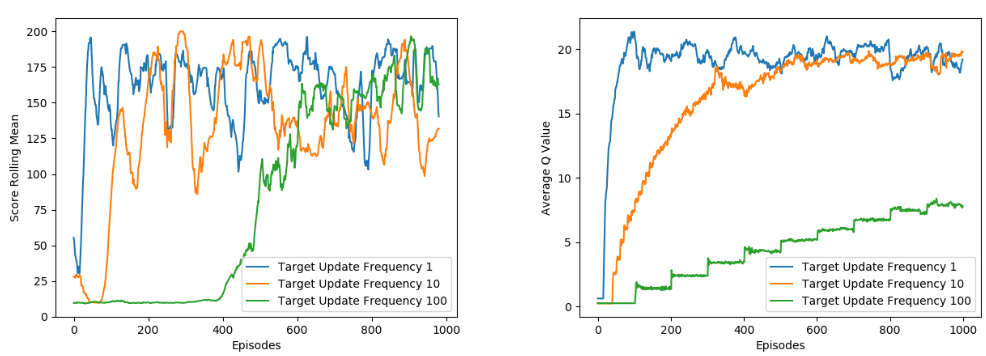

Figure 10: (Left) Rolling mean of the scores per episode for different target updates frequencies. (Right) The Average Q-value per episode for different target update frequencies.

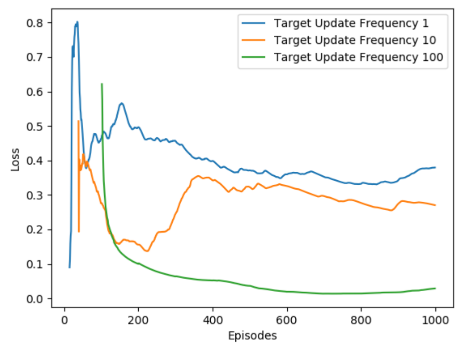

Figure 11: Average loss function for each episode for different target update frequencies.

6.5 Extension

In the extension, we try to further improve the learning performance of the model. First, this is done by introducing a decaying learning rate. This means that in the start of the learning process, when we are supposedly far from a solution, the agent will take larger steps towards the solution, decreasing the learning time. When we are getting closer to the minimum, the steps become smaller, increasing the precision of the algorithm. When introducing a decay parameter, we will alter the learning rate, shifting it upwards to counter the decay.

Further, we investigate different activation function for the hidden layer in the neural network. If several of the activation's gets below zero, many of the neurons in the network will "die" and prohibit learning, a phenomenon commonly referred to as "the dying ReLU problem" [2]. To cope with this we tried using a sigmoid and tanh activation function instead. Since we are not using multiple layers, the problem with vanishing gradients won't be significant.

7 Conclusion

Finding an optimal model that solves the CartPole problem has constitute partly of a trail-and-error procedure. This combined with the experience gathered during the investigation section of this lab, has allowed us to choose an appropriate set of hyper-parameters and model.

N hidden layers	1
N neurons	84
Discount factor	0.95
Learning rate	0.03
Decay Rate	0.0008
Memory size	10000
Activation function	Sigmoid
Update frequency	1

Table 4: Chosen hyper-parameter settings

Our best model solved the problem after 127 episodes with the chosen hyper-parameters shown in Table 4.

[2]https://en.wikipedia.org/wiki/Rectifier_(neural_networks)

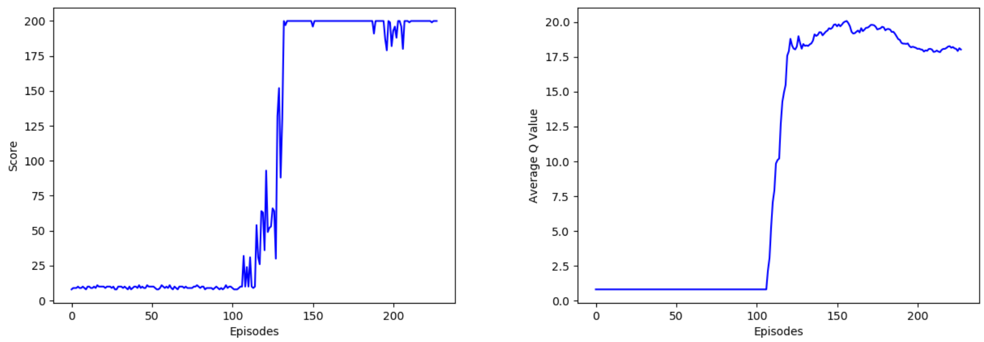

Figure 12: (Left) Scores per episode for our chosen model. (Right) The Average Q-value per episode for our chosen model

8 Appendix A

```python
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
from keras.callbacks import History

history = History()


EPISODES = 1000  # Maximum number of episodes

# PLOT VARIABLES

nr_layers = 1

#Name_of_plot = "Learning Rate"
#Name_of_plot = "Discount Factor"
#Name_of_plot = "Memory Size"
#Name_of_plot = "Target Update Frequency"
#Name_of_plot = "Default"
#Name_of_plot = "Record"
Name_of_plot = "Number␣of␣Neurons"

learning_rate = 0.03
discount_factor = 0.95
memory_size = 10000
target_update_frequency = 1
nodes = 100

# Iteration parameters
#parameter_variable = [0.05, 0.005, 0.0005]
#parameter_variable = [0.90, 0.95, 0.99]
#parameter_variable = [1000,10000,100000]
#parameter_variable = [1,10,100]
parameter_variable = [24,48,84,100]

parameter_variable = [100]

# 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, nodes_first, learning_rate, discount_
        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

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

```

```
        ##############################################################################
        ##############################################################################
        # Set hyper parameters for the DQN. Do not adjust those labeled as Fixed.

        self.discount_factor = discount_factor  # lambda
        self.learning_rate = learning_rate
        self.memory_size = memory_size

        self.epsilon = 0.02  # Fixed
        self.batch_size = 32  # Fixed
        self.train_start = 1000  # Fixed

        self.target_update_frequency = target_update_frequency

        # New added hyper parameteres
        self.nodes_first_hidden = nodes_first
        ##############################################################################
        ##############################################################################

        # 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()

        self.losses = []

        # 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(self.nodes_first_hidden, input_dim=self.state_size, activatio
                        kernel_initializer='he_uniform'))
        #model.add(Dense(24, 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, decay=0.0008
        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.
        if random.uniform(0, 1) < self.epsilon:
            action = random.randrange(self.action_size)
        else:
            Q_state = self.model.predict(state)
            action = np.argmax(Q_state)
        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-dimension
        update_target = np.zeros((batch_size, self.state_size))
# Same as above, but used for the target network
        reward = np.zeros((batch_size, 1))
        action, 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)
            reward[i] = mini_batch[i][2]
            update_target[i] = mini_batch[i][
                3]  # Allocate s'(i) for the target network array from iteration i in
            done.append(mini_batch[i][4])  # Store done(i)

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

        # 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 ep
```

```
        max_Qtarget = np.reshape(np.max(target_val, axis=1), (batch_size, 1))
        max_Qtarget[done] = 0  # Q-value of the action taken at the last state of the
        y = reward + self.discount_factor * max_Qtarget

        # Line 10 in Algo. 1

        for i in range(self.batch_size):  # For every batch
            target[i][action[i]] = y[i]  # random.randint(0,1)
        # print(target)
        ##########################################################################
        ##########################################################################

        # Train the inner loop network
        history = self.model.fit(update_input, target, batch_size=self.batch_size,
                      epochs=1, verbose=0)

        self.losses.append(history.history['loss'][0])

        return



    # Plots the score per episode as well as the maximum q value per episode, average
    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")
        pylab.show()


def multi_plot(episodes, all_scores, all_max_q_mean,all_loss,agent):

    pylab.figure(0)

    discount_factor = agent.discount_factor
    learning_rate = agent.learning_rate
    memory_size = agent.memory_size
    nodes = agent.nodes_first_hidden

    # Rolling Average
    N = 20


    for i in range(np.shape(all_scores)[0]):
        cumsum = np.cumsum(np.insert(all_scores[i], 0, 0))
        moving_avg = (cumsum[N:] - cumsum[:-N]) / float(N)
        pylab.plot(moving_avg, label= Name_of_plot + " " + str(parameter_variable[i]
    pylab.xlabel("Episodes")
    pylab.ylabel("Score Rolling Mean")
    pylab.legend()
```

```
    pylab.savefig("C:/Users/Therese/PycharmProjects/ReinforcementLearningLAB1/plots/%
    pylab.show()


    for i in range(np.shape(all_scores)[0]):
        pylab.plot(episodes, all_max_q_mean[i][:], label= Name_of_plot + " " + str(p
    pylab.xlabel("Episodes")
    pylab.ylabel("Average Q Value")
    pylab.legend()
    pylab.savefig("C:/Users/Therese/PycharmProjects/ReinforcementLearningLAB1/plots/%
    pylab.show()

    pylab.figure(1)
    for i in range(np.shape(all_scores)[0]):
        pylab.plot(episodes, all_scores[i][:], label= Name_of_plot + " " +
str(parameter_variable[i]))
    pylab.xlabel("Episodes")
    pylab.ylabel("Score")
    pylab.legend()
    pylab.savefig("C:/Users/Therese/PycharmProjects/ReinforcementLearningLAB1/plots/%
    pylab.show()

    pylab.figure(2)
    for i in range(np.shape(all_scores)[0]):
        pylab.plot(episodes, all_loss[i][:], label= Name_of_plot + " " +
str(parameter_variable[i]))
    pylab.xlabel("Episodes")
    pylab.ylabel("Loss")
    pylab.legend()
    pylab.savefig("C:/Users/Therese/PycharmProjects/ReinforcementLearningLAB1/plots/%
    pylab.show()



if __name__ == "__main__":
    all_scores = np.zeros((len(parameter_variable), EPISODES))
    all_max_q_mean = np.zeros((len(parameter_variable), EPISODES))
    all_losses = np.zeros((len(parameter_variable), EPISODES))

    for n in range(len(parameter_variable)):

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

        action_size = env.action_space.n

        # Setting iteration variable (Mainly for the plots)
        if Name_of_plot == "Learning Rate":
            learning_rate = parameter_variable[n]

        if Name_of_plot == "Memory Size":
            memory_size= parameter_variable[n]

        if Name_of_plot == "Discount Factor":
            discount_factor = parameter_variable[n]

        if Name_of_plot == "Target Update Frequency":
```

```
            target_update_frequency = parameter_variable[n]

        if Name_of_plot == "Number of Neurons":
            nodes = parameter_variable[n]

        # Create agent, see the DQNAgent __init__ method for details
        agent = DQNAgent(state_size, action_size, nodes,learning_rate,discount_factor


        print(agent.learning_rate, agent.discount_factor, agent.memory_size)
        #print("Learning Rate %f" %agent.learning_rate)

        # agent.nodes_first_hidden = nodes[n]

        # 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, losses = [], [], []
# 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
            # 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

                    loss_list = agent.losses
                    losses.append(np.mean(loss_list))

                    scores.append(score)
                    episodes.append(e)

                    print("episode:", e, "  score:", score, " q_value:", max_q_mean[
                          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)
        all_scores[n][:] = np.asarray(scores)
        all_max_q_mean[n][:] = max_q_mean.ravel()
        #print(losses)
        all_losses[n][:] = np.asarray(losses)

    multi_plot(episodes, all_scores, all_max_q_mean,all_losses, agent)
```