EL2805 Reinforcement Learning Lab 2
Deep Reinforcement Learning for Cartpole

Sri Datta Budaraju
budaraju@kth.se
971029-2450

Aniss Medbouhi
medbouhi@kth.se
950907-1578

A Formulation of the RL problem

We propose the following formulation:

State space $\mathcal{S}$

At every time we can observe both the cart position x (w.r.t. the centre) and the pole angle θ, as well as their derivatives $\dot{x}$ and $\dot{\theta}$ so by noting L the distance between the centre and one side, we have:
$s_t = (x, \dot{x}, \theta, \dot{\theta}) \in \mathcal{S} = [-L, L] \times \mathbb{R} \times [-90, 90] \times \mathbb{R}$

Action space $\mathcal{A}$

At every time a force of magnitude 1N can be applied on either side of the cart so:
$a_t \in \mathcal{A} = \{-1, 1\}$

Rewards $\mathcal{R}$

$r_t(s_t, a_t) = 1$ if $s_t \in \mathcal{S} = [-2.4, 2.4] \times \mathbb{R} \times [-12.5, 12.5] \times \mathbb{R}$ and $r_t(s_t, a_t) = 0$ otherwise.

Time-horizon and objective

We have a discounted finite horizon, we want to maximise $\mathbb{E}\left\{\sum_{t=0}^{T} \lambda^t r_t\left(s_t, a_t\right)\right\}$ with $T = min\{$200; the first time t where $r_t(s_t, a_t) = 0\}$.

Transition probabilities $\mathcal{P}$

These encode the dynamics of the system... In this lab we will try to learn it directly from the environment.

Standard RL versus deep RL

We notice here that the state-space $\mathcal{S}$ is infinite because continuous. So in order to solve it we need to discretise it, but this leads to a finite state-space with too many states. That is why standard RL methods cannot be applied to solve it! So we will use deep RL methods to approximate (state, action) value functions through neural networks.

B Description of the code

B.1 Description of the $main$

First the $main$ generates the Cartpole-v0 environment object from the gym library and gets the state and action sizes to create the agent which is an object of the class DQNAgent.

After that, it collects the test states for plotting the Q values using a uniform random policy: while the terminal state s_T is not reached it takes a random action from $\mathcal{A}$, and when it is reached it restarts the environment to $s_1 = (0, 0, 0, 0) \in \mathcal{S}$ and plays randomly again etc.

Then comes the DQN as explained in next section. We play the 1000 episodes. For each episode, after initializing the environment and computing the Q values from $agent.model.predict()$, while the terminal state is not reached we take an action according to $agent.get_action()$ and go one step in environment and save the sample (s_t, a_t, r_t, s_{t+1}) to the replay memory. We also train the model and update the score before. Then when the terminal state of the current episode is reached, if we have done $agent.target\_update\_frequency$ number of episodes without updating the target network then we update it. Moreover at the end of each episode we plot the scores, and if the mean scores of the last 100 episodes is bigger than 195 we stop to train.
Finally we plot the episodes, the scores and the mean of the Q values (for all the episodes).

B.2 Brief description of the behavior of each function

__init__(self, state_size, action_size)

This is the constructor for the class DQNAgent.

build_model(self)

This is for building a Neural Network model.

update_target_model(self)

This is for updating the target model to be the same with model every $agent.target\_update\_frequency$ times, in order to "slow down" the evolution of the target so it can be tracked while θ is constantly updated.

get_action(self, state)

This is for getting the action according to the ϵ-greedy policy with respect to Q.

append_sample(self, state, action, reward, next_state, done)

This function allows us to save the sample (s_t, a_t, r_t, s_{t+1}) to the replay memory.

train_model(self)

This function allows us first to sample $self.batch\_size$ experiences (s_i, a_i, r_i, s_{i+1}) from the replay memory. Then it generates the target values for training the inner loop network using the network model, and the target values for training the outer loop target network. After that comes the Q-Learning part to get the maximum Q value at s_{t+1} from the target network. Finally the inner loop network is trained.

plot_data(self, episodes, scores, max_q_mean)

This is just to plot the Average Q Values and the Scores in function of the episodes.

C DQN pseudo-code and correspondence with the code

Here are the correspondences with the provided code:

The line 1 corresponds to the iterations of the while loop line 179 in the code. It ends when the terminal state is reached because the time horizon is finite.
The line 2 is for computing π_t the ϵ -greedy policy w.r.t. Q_θ, and the line 3 is to take the action according to this policy computes and to observe the reward and the next state. So they correspond to the line 184 in the code which calls the function $agent.get\_action(state)$ implemented line 72 in the code: takes a random (uniform probability) action with probability ϵ to explore, and the

Algorithm 1: DQN pseudo-code

```
Input   : initialized $\theta$ and $\phi$, replay buffer $B$, initial state $s_1$
Output : $Q$ value function
1  for t = 1 to T (until reaching a terminal state) do
2      compute $\pi_t$ the $\epsilon$ -greedy policy w.r.t. $Q_\theta$
3      take action a$_t$ according to $\pi_t$, and observe $r_t, s_{t+1}$
4      store $(s_t, a_t, r_t, s_{t+1})$ in $B$
5      sample k experiences $(s_i, a_i, r_i, s'_i)$ from $B$
6      for i = 1 to k (compute using the target net. with weights $Q_\phi$ do
7          if episode stops in $s'_i$ then
8              $y_i = r_i$ else
9                  $y_i = r_i + \lambda \max_b Q_\phi (s'_i, b)$ end
10             end
11         end
12         update the weights $\theta$ (using back prop.) as:
13         $\theta \leftarrow \theta + \alpha (y_i - Q_\theta (s_i, a_i)) \nabla_\theta Q_\theta (s_i, a_i)$
14         outside the for loops
15         every $C$  steps:  $\phi \leftarrow \theta$
16
```

$\arg\max_{b\in A_s} Q_\theta^{(t)}(s, b)$ with probability $1 - \epsilon$ to exploit. And the observation of the reward and the next state corresponds to the line 185 in the code.
The line 4 is to save the sample (s_t, a_t, r_t, s_{t+1}) to the replay memory which corresponds to the line 189 in the code.
Then lines 5 to 13 correspond to the line 191 in the code which calls the function $agent.train\_model()$. So more precisely:
The line 5 which samples k experiences from B corresponds to the lines 97 to 102 in the code and $k = self.batch\_size$ is the number of episodes. The for-loop line 6 corresponds to the for-loop line 113 in the code. But the "compute using the target net. with weights Q_ϕ" corresponds to the lines 104 and 105 in the code. Then lines 7 to 11 correspond to the incomplete code about about Q-learning. Line 7 is to say that when the terminal state is reached (that is to say when the current episode is finished), the target should receive only the reward (line 8). Otherwise (line 10) it receives $r_i + \lambda \max_b Q_\phi (s'_i, b)$. The lines 12 and 13 are about updating the weights θ using back propagation, which corresponds to line 119 in the code.
Finally the line 15 is about updating the target network at the end of every episode every C steps, so it corresponds to lines 197 and 198 in the code with $C = agent.target\_update\_frequency$.

D Network Architecture

The model is sequential, so it is a linear stack of layers. The input shape is $self.state\_size$. There is one fully connected hidden layer with a ReLU as activation function and a units number equal to 16, and one output layer with a linear activation function and an output shape equal to the number of possible actions which is 2.
The learning process is configured with a mean squared error as a loss function, and Adam as an optimizer with a learning rate equal to $self.learning_rate$.

E ϵ-greedy Implementation

Please refer the appendix

F Complete $train\_model$

Please refer the appendix

G Impact of Architecture Parameters

From this section the experiments are illustrated using graphs as follows. To the left are the KDEs of the average Q value and to the right is the KDEs of the score. We interpret our results using KDE since the distribution of the values are more informative than the raw training graphs. Above that the training plots are not very distinguishable from one another and is hard to derive anything by looking at it. Nothing interesting has been observed in the training plots for most experiments other than varying the update frequency.

In the figure 1 we vary the **number of neurons** in hidden layer while keeping the other parameters as it is. Though there is no significant difference by just changing the number of neurons, We can observe from that the Q values are slightly higher for network with higher neurons. And similar but inconsistent trend could be seen in the scores as well.

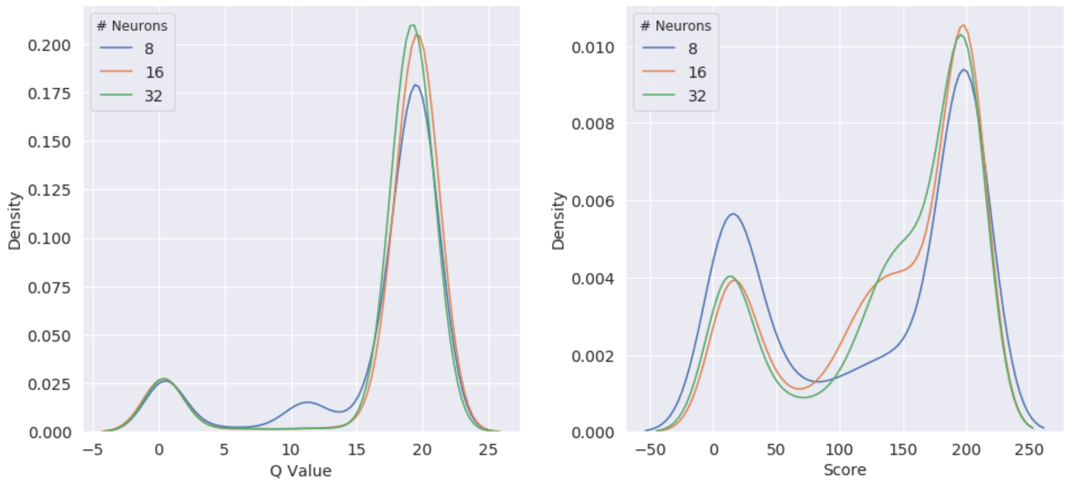

Figure 1: Impact of varying number of Neurons

In the figure 2 we vary the **number of layers** while keeping the other parameters as it is. We can observe from that the both Q values and score are much higher for network with 3 layers. Though model with 1 layer seem to be equivalent to one with 2 layers, its safe to say more layers might improve the performance since 3 layer model is performing significantly better than the other 2.

H Investigating Other Hyperparameters

In the figure 3 we vary the **discount factor** while keeping the other parameters as it is. We can see that higher discount 0.95 has better result than 0.7. And we wanted to check what would happen if discount is 0.5. i.e not favoring the best action. As expected the Q value has not improved and scores are pretty low.

In the figure 4 we vary the **learning rate** while keeping the other parameters as it is. As we know that lower learning rate takes more time to converge and higher learning rate might overshoot the minima. Hence we train with reduce the default learning rate 5 times to 0.001 and also increase it 5 times to 0.025 to see if we can observe such a pattern. And in the figure 4 we can see the same in both the plots.

In the figure 5 we vary the **size of the memory** while keeping the other parameters as it is. Here we initially train using 1000 memory size, then half and double it. And we also wanted to see how could it utilize a very small memory so we also train using a memory of size 200. The results are interesting. We see that large memory have better scores than medium and very small but lower Q values. This could be the case that samples from the buffer are seen less frequently by the model but since it learns diverse experiences it performs better. Where are 500 memory size does better both on

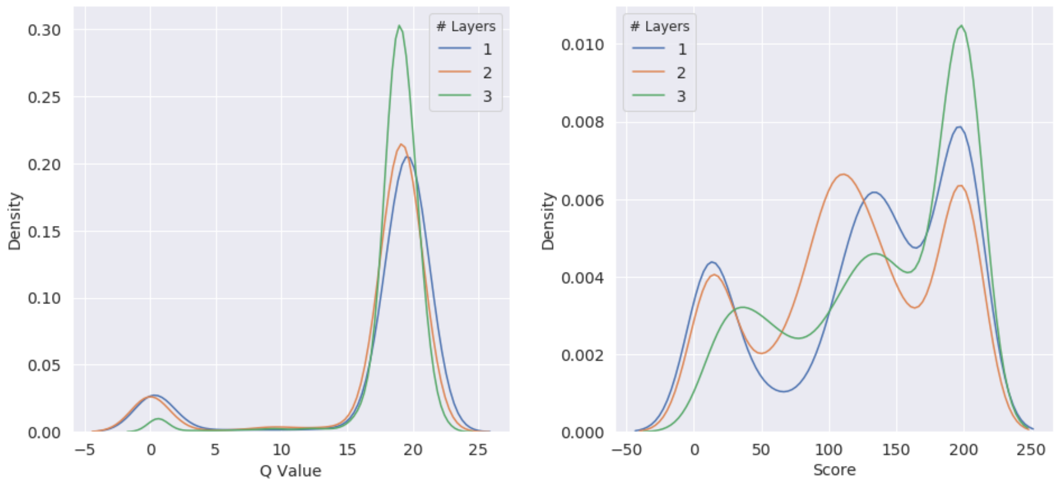

Figure 2: Impact of varying number of Layers

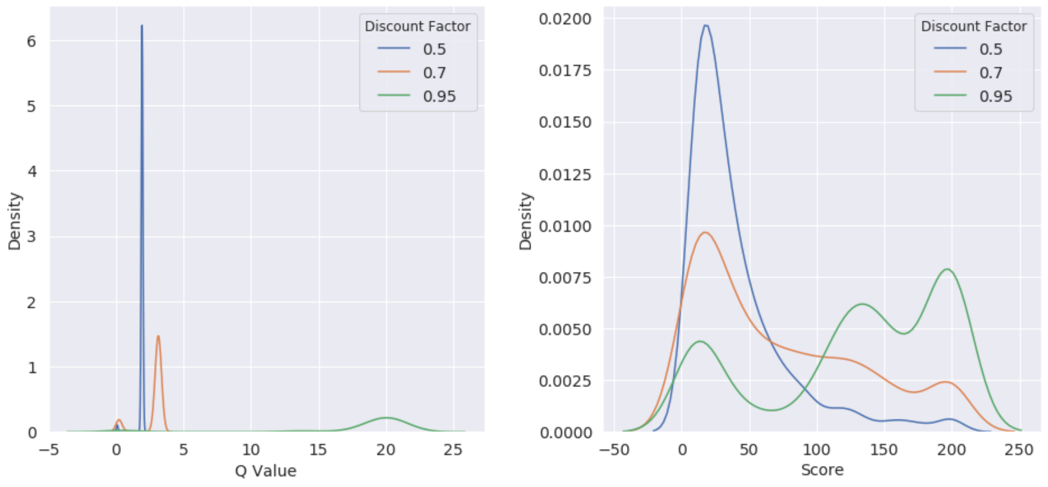

Figure 3: Impact of varying discount factor

Q values and the scores. Where as, the 200 memory size is performing well on Q values but not as good on the score. Could be interpreted as overfitting to the experience.

I Investigating the Impact of Target Model Update Frequency

In the figure 6 we vary the **update frequency** while keeping the other parameters as it is. As we know that the frequency actually is the time period in which the target model is updated. The longer time period the older the target model. When the frequency is 1, we update the target network every episode. This could make the training unstable due to frequent change in the target value. This could be seen in figure 7, the graph is more unstable compared to other training plots. We can directly see the update pattern in this figure. The Q values are similar for the time period where the target is not updated. The longest time period here is 100 and we can see clear steps in the graph. From figure 6 we can see that the higher the frequency the worse the performance since, the target values are outdated. This could be seen that the model is trying to learn the wrong actions.

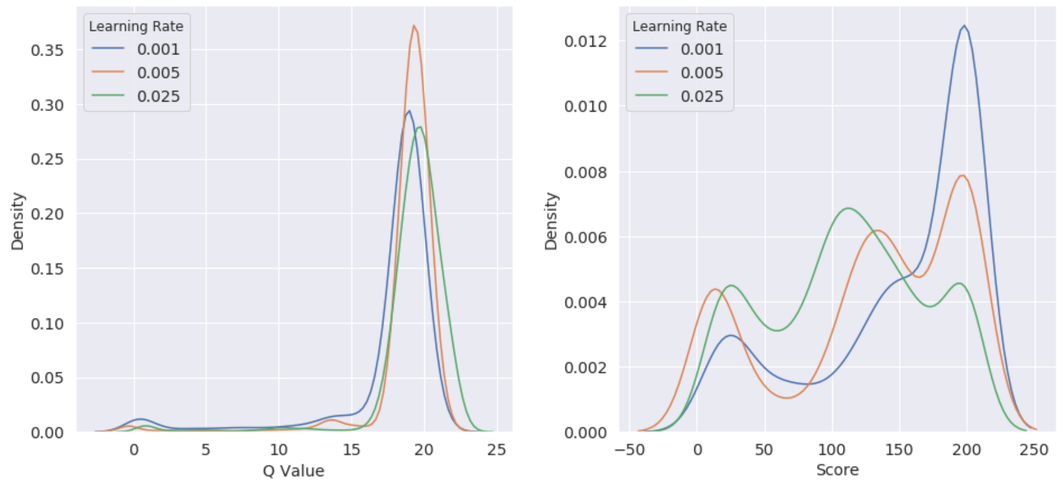

Figure 4: Impact of varying learning rate

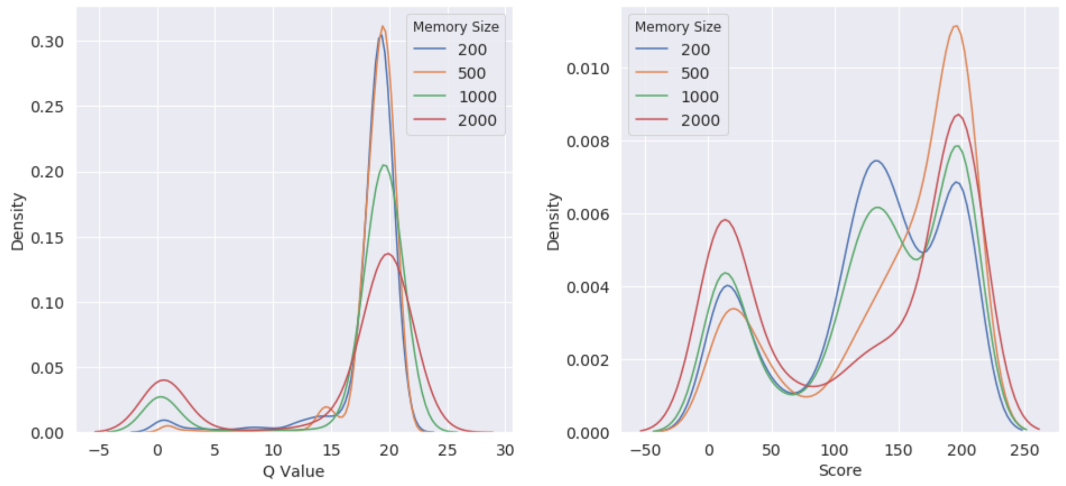

Figure 5: Impact of varying memory size

J Solving the Problem

The figure 8 illustrates the stats in Q values and Score for the model that solved the cart-pole problem.

The following model solved the problem after 59 episodes of training. The hyper parameters are chosen based on the impacts observed in the above experiments.

Neurons 32 - Since it has more capacity to learn and we have observed more to be better.
Layers 3 - Outperformed 1 and 2 in the experiments
Learning Rate 0.001 - Since we need average score to be >195, we need it to be consistent and stable
Discount Factor 0.99 - We have seen that higher discount factor improves performance significantly
Memory Size 1000 - Have got good results with other size as well.
Update Frequency 1 - It outperformed others in the experiments

K Appendix

The code is below.

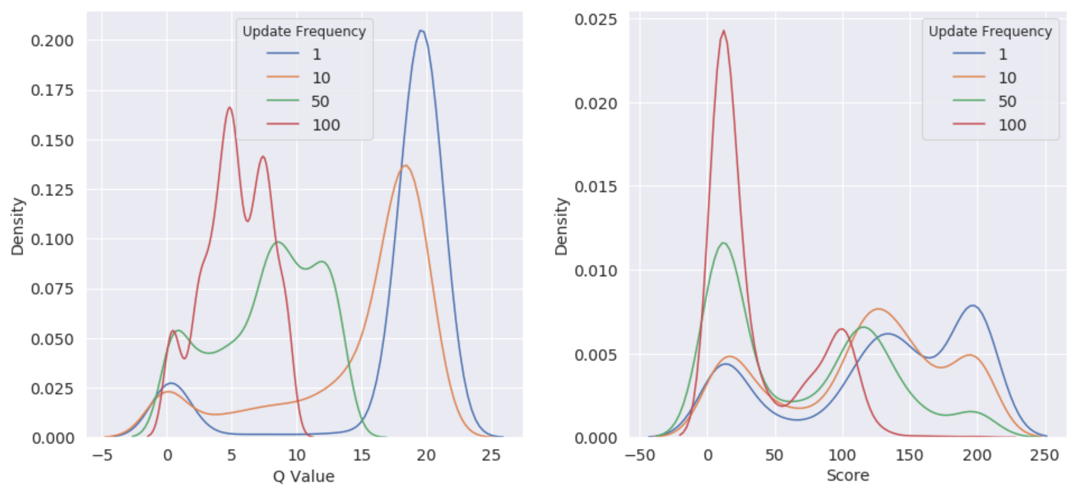

Figure 6: Impact of update frequency

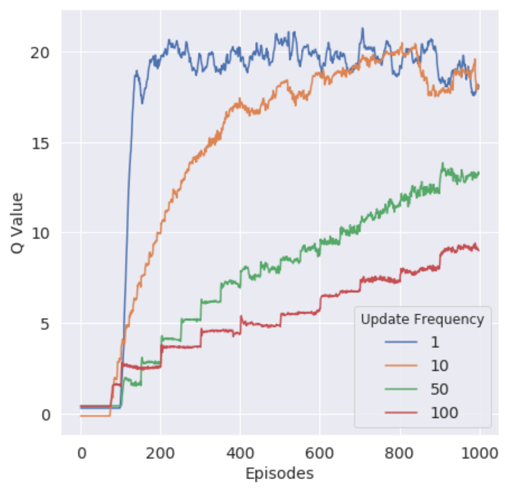

Figure 7: Impact of update frequency - Training Plot

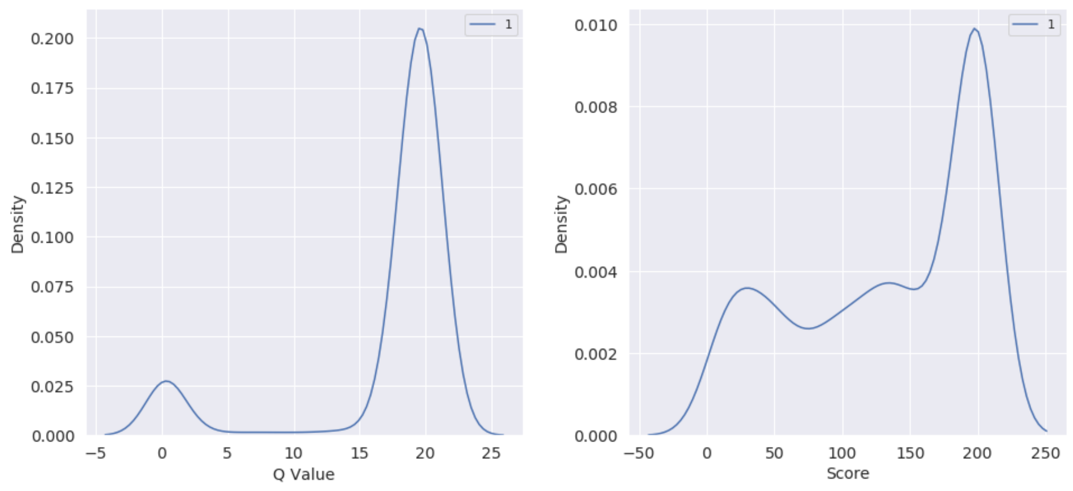

Figure 8: Winning Model

In [ ]:
```
### DQN Code ###

import sys
import gym
import pylab
import random
import numpy as np

from argparse import ArgumentParser
from time import time
from collections import deque

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 netw
ork
class DQNAgent:
    #Constructor for the agent (invoked when DQN is first called in m
ain)
    def __init__(self, state_size, action_size, hparams):
        self.check_solve = True    #If True, stop if you satisfy solu
tion confition
        self.render = False        #If you want to see Cartpole learn
ing, then change to True

        #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 labele
d as Fixed.
        self.discount_factor = hparams.discount
        self.learning_rate = hparams.lr
        self.epsilon = 0.02 #Fixed
        self.batch_size = 32 #Fixed
        self.memory_size = hparams.memory
        self.train_start = self.memory_size #Fixed
        self.target_update_frequency = hparams.frequency
        self.neurons = hparams.neurons
        self.layers = hparams.layers

        #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 de
fined 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.
    def build_model(self):
        model = Sequential()
        model.add(Dense(self.neurons, input_dim=self.state_size, acti
vation='relu',
                        kernel_initializer='he_uniform'))

        for _ in range(self.layers-1):
            model.add(Dense(self.neurons, 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_rat
e))
        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):
        Q = self.model.predict(state)
        max_a = np.argmax(Q)
        action_probs = np.dot([1]*self.action_size, \
                       self.epsilon/self.action_size)
        action_probs[max_a] += (1 - self.epsilon)
        action = np.random.choice(range(0,self.action_size), p=action
_probs)
        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) #Uniforml
y 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 gro
w 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 t
he 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 va
lues for training the inner loop network using the network model
        target_val = self.target_model.predict(update_target) #Genera
te the target values for training the outer loop target network

        #Q Learning: get maximum Q value at s' from target network
        for i in range(self.batch_size): #For each in batch
            if done[i]:
                target[i][action[i]] = reward[i]
            else:
                target[i][action[i]] = reward[i] + \
                self.discount_factor * np.max(target_val[i])

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

    #Plots the score per episode as well as the maximum q value per e
pisode, averaged over precollected states.
    def log_data(self, episodes, scores, max_q_mean):

        name = f'_best_{self.target_update_frequency}'

        pylab.figure(0)
        pylab.plot(episodes, max_q_mean, 'b')
        pylab.xlabel("Episodes")
        pylab.ylabel("Average Q Value")
        pylab.savefig(f"cartpole/plots/qvalues{name}.png")
        np.savez(f"cartpole/pickles/qvalues{name}.npz", max_q_mean)

        pylab.figure(1)
        pylab.plot(episodes, scores, 'b')
        pylab.xlabel("Episodes")
        pylab.ylabel("Score")
        pylab.savefig(f"cartpole/plots/scores{name}.png")
        np.savez(f"cartpole/pickles/scores{name}.npz", scores)

if __name__ == "__main__":

    parser = ArgumentParser()

    parser.add_argument("--neurons", type=int, default=16)
    parser.add_argument("--layers", type=int, default=1)

    parser.add_argument("--discount", type=float, default=0.95)
    parser.add_argument("--memory", type=int, default=1000)
    parser.add_argument("--lr", type=float, default=0.005)

    parser.add_argument("--frequency", type=int, default=1)

    hparams = parser.parse_args()

    #For CartPole-v0, maximum episode length is 200
    env = gym.make('CartPole-v0') #Generate Cartpole-v0 environment o
bject 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, hparams)

    #Collect test states for plotting Q values using uniform random p
olicy
    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

    mean_score = 0
    max_mean_score = 0
    scores, episodes = [], [] #Create dynamically growing score and e
pisode 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 envi
ronment
            action = agent.get_action(state)
            next_state, reward, done, info = env.step(action)
            next_state = np.reshape(next_state, [1, state_size]) #Res
hape next_state similarly to state

            #Save sample <s, a, r, s'> to the replay memory
            agent.append_sample(state, action, reward, next_state, do
ne)
            #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 networ
k
                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 agent.check_solve:
                    print("episode:", e, "  score:", score," q_valu
e:", max_q_mean[e],"  memory length:",
                      len(agent.memory), " mean_score:", mean_score)
                else:
                    print("episode:", e, "  score:", score," q_valu
e:", max_q_mean[e],"  memory length:",
                      len(agent.memory))

                # if the mean of scores of last 100 episodes is bigge
r than 195
                # stop training
                mean_score = np.mean(scores[-min(100, len(scores)):])
                if agent.check_solve:
                    if mean_score > max_mean_score:
                        max_mean_score = mean_score
                    if np.mean(scores[-min(100, len(scores)):]) >= 19
5:
                        print("solved after", e-100, "episodes")
                        agent.log_data(episodes,scores,max_q_mean[:e+
1])
                        sys.exit()

    agent.log_data(episodes,scores,max_q_mean)

    if agent.check_solve:
        print(f"Max Mean Score for this config: {max_mean_score}")
```

In [ ]:
```
### Visualization Code ###

import os
import numpy as np
import seaborn as sns
from matplotlib import pyplot as plt

# all hyperparameters and metrics logged
cats = {'qvalues': "Q Value", 'scores':"Score"}

hparams = {'layers':'# Layers', 'discount':'Discount Factor',\
           'freq':'Update Frequency', 'lr':'Learning Rate',\
           'neurons': "# Neurons", 'mem': "Memory Size"}
defaults = {'layers':1, 'discount':0.95,\
           'freq':1, 'lr':0.005,\
           'neurons': 16, 'mem': 1000}

# to plot Q values and scores
def plot(hparam, cat):
    _ = plt.rcParams["figure.figsize"] = (7,7)
    _ = plt.tick_params(axis='both', which='both', labelsize=14)

    files = {}
    for file in os.listdir("cartpole/pickles/"):
        if hparam in file and cat in file:
            # Get the parameter value
            val = '.'.join(file.split(hparam+'_')[1].split('.')[0:-1
])

            # Default key is arr_0, arr_1, ....
            files[val] = np.load(f'cartpole/pickles/{file}')['arr_0']
        elif "layers_1" in file and cat in file:
            val = str(defaults[hparam])
            files[val] = np.load(f'cartpole/pickles/{file}')['arr_0']

    # sort the labels
    sorted_files = dict(sorted(files.items(), key=lambda x: float(x[0
])))
    sns.set()
    _ = plt.xlabel(cats[cat], fontsize=14)
    _ = plt.ylabel("Density", fontsize=14)

    for x in sorted_files.keys():
        _ = sns.distplot(files[x], label=x,
                         hist=False, bins=200)
    _ = plt.legend(title=hparams[hparam], fontsize=14)
    _ = plt.savefig(f'cartpole/vizs/{hparam}_{cat}.png', bbox_inches=
'tight')
    _ = plt.clf()
    print(f"Done {hparam}, {cat} - {sorted_files.keys()}")

for cat in cats.keys():
    for hparam in hparams.keys():
        plot(hparam, cat)
```