EL2805 Computer Lab 2: *Deep Q Learning*

Deepika Anantha Padmanaban
19950303-4408
deap@kth.se

Styliani Katsarou
900331-7006
stykat@kth.se

1 Problem 1: Deep Reinforcement Learning for Cartpole

1.1 Formulation of the RL problem

State Space: $s_t = (x, \dot{x}, \theta, \dot{\theta}) \in S = [-d, +d] \times \mathbb{R} \times [-\pi/2, \pi/2] \times \mathbb{R}$

where d is the distance between the centre and one of the walls

Actions: A force of magnitude $1N$ can be applied on either side of the cart so there are two actions, one for the right and one for the left direction of the force: $a_t \in \mathcal{A} = \{-1, 1\}$

Time Horizon and Objective: Discounted infinite horizon $\mathbb{E}\{\sum_{t=0}^{\infty} \lambda^t r_t(s_t, a_t)\}$

Rewards: $r(s_t, \cdot) = 1$ if $\theta \in [-12.5°, +12.5°]$, and $x \in [-2.4, +2.4]$. Zero otherwise.

Every episode of this task is a repeated attempt to balance the pole. Since the reward for each time that θ is kept within $12.5°$ from the vertical position and the magnitude of x is less than 2.4m, the return is the number of time steps until failure or until the end of the episode which is equal to 200 time steps. Successful balancing an episode means that the total score will be 200.

$x, \dot{x}, \theta, \dot{\theta}$ are continuous variables, so the number of possible combinations of these four variables is too large, leading to a state space so large that it is computationally inefficient or infeasible to iteratively compute and update q-values for each state-action pair, due to computational resources and time it may take. Rather than using methods to directly compute q-values and find the optimal q-function, we can instead use function approximation techniques, and that is where deep learning steps in: by providing deep neural networks as efficient function approximators.

1.2 Brief description outlining the steps taken in main

First we create the Cartpole-v0 environment object from gym library:

```
env = gym.make('CartPole-v0')
```

We get the state and action sizes out of this environment:

```
state_size = env.observation_space.shape[0] # outputs 4
action_size = env.action_space.n # outputs 2
```

next we create a DQNSolver agent, with an observation space (possible state values) and an action space (possible actions that can be performed)

```
agent = DQNAgent(state_size, action_size)
```

we now pick 10.000 states. Note that these are sequential, but generated by taking random actions. Esentially we want to end up having an average of maximum q value correpsponding to each episode and that is what will be ploted in the end.

We will be using our trained model for this particular line:

```
tmp = agent.model.predict(test_states)
```

This way, we will be able to check whether our model improves over episodes. This happens because the model weights are updates in the inner loop during training (the one loop that iterates over time steps) to reduce the loss between the target and the q values predicted by the current model:

```
test_states = np.zeros((agent.test_state_no, state_size))
```

and we fill the matrix up, with states chosen randomly:

```
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)  # choose a random action out of the two
        next_state, reward, done, info = env.step(action)  #
        next_state = np.reshape(next_state, [1, state_size])
        test_states[i] = state
        state = next_state
```

we initialize max q and max q mean :

```
max_q = np.zeros((EPISODES, agent.test_state_no))
max_q_mean = np.zeros((EPISODES,1))
```

we now initialize the two lists that we will be using for our plots:

```
scores = []
episodes = []
```

The next lines of code are going to fill these lists up. The steps will be described with comments inside the code snippet:

```
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]]
    # SO HERE WE USED OUR TRAINED MODEL TO PREDICT THE Q VALUES (this is the tmp).
    # RECALL THAT WE ARE DOING THIS FOR THE SET OF RANDOM STATES CREATED BEFORE.
    # SO FOR EACH OF THE 10.000 STATES WE TAKE THE MAXIMUM OF THE TWO Q VALUES IN THE tmp MATRIX = THAT WILL GIVE US max_q (SHAPE IS ( episodes,10.000)
    # AND THEN FOR EACH EPISODE WE TAKE THE MEAN OF THE 10.000 VALUES = THAT WILL GIVE US max_q_mean (shape is (episodes,1))
    # AND THIS IS WHAT WE WILL EVENTUALLY PLOT.
    #Compute Q values for plotting
    tmp = agent.model.predict(test_states) # 10.000 x 2
    max_q[e][:] = np.max(tmp, axis=1) #  episodes x 10.000
    max_q_mean[e] = np.mean(max_q[e][:]) #  episodes x 1

    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) # EXPLORE OR EXPLOIT
        #EXECUTE SELECTED ACTION IN THE GYM EMULATOR AND OBSERVE REWARD AND NEXT STATE. ALSO WHETHER WE ARE DONE OR NOT. (FAILURE OR  NOT)
        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) # STORE EXPERIENCE IN THE REPLAY MEMORY
        #Training step
        agent.train_model() # HERE WE USE OUR TWO NETWORKS.
        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)
```

1.3 Explanation of DQN pseudo-code and association with corresponding parts of the code provided

We will be using the pseudocode depicted in Fig.1:

1. **Initialization.** θ and ϕ, replay buffer B, initial state s_1
2. **Iterations:** For every $t \geq 1$,
   compute π_t the ϵ-greedy policy w.r.t. Q_θ
   take action a_t according to π_t, and observe r_t, s_{t+1}
   store (s_t, a_t, r_t, s_{t+1}) in B
   sample k experiences (s_i, a_i, r_i, s_i') from B
   for $i \in [1, k]$: compute using the target net. with weights ϕ

$$y_i = \begin{cases} r_i & \text{if episode stops in } s_i' \\ r_i + \lambda \max_b Q_\phi(s_i', b) & \text{otherwise} \end{cases}$$

   update the weights θ (using back prop.) as:

$$\theta \leftarrow \theta + \alpha(y_i - Q_\theta(s_i, a_i))\nabla_\theta Q_\theta(s_i, a_i)$$

   every C steps: $\phi \leftarrow \theta$

Figure 1: Pseudocode for DQN

Note: The rest of this section mentions the lines of code that correspond to the pseudocode depicted above.

Thus, the code snippets mentioned below cannot be used in the exact order mentioned. They are put in this order just to demonstrate the lines of code that correspond to the aforementioned pseudocode, as part of the answer of question (c).

We initialize of and :

```
self.discount_factor = 0.95
self.learning_rate = 0.005
self.epsilon = 0.02 #Fixed
self.batch_size = 32 #Fixed
self.memory_size = 1000
self.train_start = 1000 #Fixed
self.target_update_frequency = 1
self.model = self.build_model()
self.target_model = self.build_model()
self.update_target_model()
```

initialization of replay memory:

```
self.memory = deque(maxlen=self.memory_size)
```

Initialization of state 1: we find this line of code in the main. We also do the essential reshaping:

```
state = env.reset()
state = np.reshape(state, [1, state_size])
```

Iterations (Specifically, line 1 iterates through episodes and line 11 iterates over time steps):

```
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:
            ...
```

Compute π_t, -greedy policy:

```
action = agent.get_action(state)
```

We then execute the selected according to π_t action in the emulator (env.step) and observe r_t and s_{t+1}:

```
next_state, reward, done, info = env.step(action)
```

Store s_t, a_t, r_t, s_{t+1} in the replay memory:

```
agent.append_sample(state, action, reward, next_state, done)
```

Sample k experiences from replay memory: This is part of the "agent.train model" function but the sampling explicitly takes place in the following line:

```
mini_batch = random.sample(self.memory, batch_size)
```

for i in $[1, k]$:

```
for i in range(self.batch_size):
```

compute using the target net with weights ϕ: if episode stops at s_{t+1}: $y_i = r_i$

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

otherwise:

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

update the weights, θ, using back-prop:

```
self.model.fit(update_input, target, batch_size=self.batch_size,
                        epochs=1, verbose=0)
```

every c steps update the target network's ϕ, with θ parametres learnt so far.

The corresponding line can be found in the main:

```
if e % agent.target_update_frequency == 0:
    agent.update_target_model()
```

f the update frequency is 1, then the update will be conducted after each episode.

1.4 Brief explanation the layout of the given neural network model

The code snippet that contains the network is:

```
def build_model(self):
        model = Sequential()
        model.add(Dense(16, input_dim=self.state_size, activation='
    relu',
                        kernel_initializer='he_uniform'))
        model.add(Dense(self.action_size, activation='linear',
                        kernel_initializer='he_uniform'))
        model.summary()
        model.compile(loss='mse', optimizer=Adam(lr=self.learning_rate
    ))
        return model
```

The Sequential model used here is a linear stack of layers.

We use the .add method to add layers. In detail:

1st layer: First we add a regular densely-connected NN layer of 16 units. This kind of layers can support the specification of their input shape via the argument input dim, so we specify it to be equal to the number of states which is 4. We specify the desired activation of this first layer via the argument activation=relu and initialize the weights using kernel initializer='he uniform. This initializer draws samples from a uniform distribution within [-limit, limit] where limit is sqrt(6 / fan in) where fan in is the number of input units in the weight tensor.

*2nd layer:*This is going to be the second and last layer of this simple network, consisting of 2 units (as many as the actions we can take out of the state that was fed into the network, that is equal to self.action size), using a linear activation and the same initializer as before.

1.5 epsilon-greedy policy code in the get action function

the ".get action" function that returns the next action according to the epsilon greedy policy is as follows:

```
def get_action(self, state):
    random_int = np.random.binomial(1, (1 - self.epsilon))
    if random_int == 1:
        qs = self.model.predict(state) #model is going to give me two
    q values, one for each action that can be taken from state.
        action = np.argmax(qs)
    elif random_int == 0:
```

```
        action = random.randrange(self.action_size)
    return action

```

1.6 train model function

The missing part of the code is:

```
 #Q Learning: get maximum Q value at s' from target network

for i in range(self.batch_size): #For every batch
    if done[i]:
        target[i][action[i]] = reward[i]
    else:
        target[i][action[i]] = reward[i] + self.discount_factor * (np.
    max(target_val[i]))

```

1.7 The impact of the model on training by adjusting the number of nodes and layers

As a metric of documenting the agent's performance, we used the average q-value over the 1000 episodes. We increased the number of nodes, varying from 8 to 128, and used up to 4 number of hidden layers. Based on the average q-value, we conclude that the performance of the agent improves by increasing the number of nodes when we only use one layer. When using two hidden layers, the performance of the agent gets better if the layers consist of different amounts of nodes. This happens because when the layers consist of the same amount of nodes, there is not much variation in the representation learnt by the layers. By increasing the number of hidden layers, the performance of the agent is improving up to a certain point. For more than 3 layers the performance is getting worse, possibly because of the big amount of hyper-parameters such a large network requires. The aforementioned conclusions can be derived by the results shown in 2.

#	Experiment	Resut
1	1layer 5 nodes	the total average is [16.85993438]
2	1layer 16 nodes	the total average is [17.00132879]
3	1layer 8 nodes	the total average is [17.45639793]
4	1layer 32 nodes	the total average is [18.02162986]
5	1layer 64 nodes	the total average is [18.79226266]
6	2layers of 16 and 8 nodes	the total average is [17.66166594]
7	2layer of 32 and 32 nodes	the total average is [16.77093529]
8	3layer 20,16,13 nodes	the total average is [17.76393154]
9	3layers of 32 16 and 8 nodes	the total average is [17.66173578]
10	3layers of 64 48 32	the total average is [18.55814505]
11	3layers of 128 100 64	the total average is [18.69447713]
12	4layers of 64,32,16,8 nodes	the total average is [16.90687171]

Figure 2: Average q-value for different model structures

1.8 Adjusting the discount factor,learning rate and memory size

Discount factor: The discount factor accounts for how much the actions of the agent in the future states affect the q-value of the current state. Making the discount factor too low(say, closer to zero) would essentially mean that the future performance of the agent is not taken into consideration at all and thus, we perform a greedy action, where the agent fails continuously as it doesn't learn with

keeping the future in mind. While having the discount factor closer to 1, would result to the future having a higher effect on the performance of the agent in the current state and thus, is better and learns and converges faster.

discount factor	Model	Result
0.1	1x64	the total average is [1.04702895]
0.5	1x64	the total average is [1.78726891]
0.99	1x64	the total average is [86.15308617]
0.5	3layers of 128 100 64	the total average is [1.78446936]
0.99	3layers of 128 100 64	the total average is [99.4431931]

Figure 3: Average q-value for discount factors

Learning rate: The learning rate decides the rate of convergence of the network. So, there is a trade-off in terms of number of episodes and learning rate for finding an optimal function approximation. Very low learning rates can make the network learn really slow, which will require more number of episodes for convergence. High learning rate also leads to divergence of the network. So, a learning rate in-between works better. For a high learning rate of 0.1, there is an average q-value of 147.84704152 over all the 1000 episodes, but the fluctuations of the scores is big. But, the fluctuations with learning rate 0.001 remained smaller when compared to the others. Based on these observations, we decided to proceed with the learning rate of 0.001, with increased number of episodes. **Memory**

learning rate	Model	Result
0.001	3layers of 128 100 64	the total average is [93.84994726]
0.005	3layers of 128 100 64	the total average is [99.4431931]
0.01	3layers of 128 100 64	the total average is [108.55568023]
0.1	3layers of 128 100 64	the total average is [147.84704152]

Figure 4: Average q-value for different learning rate

Size: An experience replay memory is used to store the agent's experiences at each time step, that is the current state of the environment, the action taken from that state, the reward given to the agent as a result of the previous state-action pair and the next state of the environment. Deep Q-learning involves two steps - action and learning. The actions that have been taken as the network learns is stored in a memory of limited size N, which stores only the last N actions taken. So, it essentially means that the size of the memory has a direct impact on the experience that the agent holds about the actions it took while learning. We expect the performance to be bad for very high and very low memory size, because in the former case, the agent doesn't update the memory with the experiences of the latest learnt model until the memory is filled, while in the latter case, it doesn't hold much experience at all. So, we expected it to be better if the memory size is medium and the experiment results turn out to be the same, with 2000 being the best in terms of average Q-value and also in terms of the less frequent fluctuations in the score, denoting a steady learning.

memory	Model	Result
100	3layers of 128 100 64	the total average is [0.46099559]
500	3layers of 128 100 64	the total average is [-0.14993168]
2000	3layers of 128 100 64	the total average is [90.33397961]
5000	3layers of 128 100 64	the total average is [106.68559647]

Figure 5: Average q-value for different memory size

1.9 Impact of target update frequency on training

The target update frequency mentions when we want our target model to be updated by the current network that's learning. The target model is what the current network would chase to approximate. But, initially both are random and we use a random network to provide targets for the learning model, but as the model learns, we are expected to update the target model with the learnt model, so the target we chase become more meaningful. So, its important to have an update that can frequently change make the target network more meaningful for imitating.Hence, we expected that a low value of target update frequency(which is essentially, frequent updates of the target network) should yield better performance and the results we receive also show the same.

frequency	Result
2	the total average is [81.58922079]
5	the total average is [57.15229549]
20	the total average is [21.00435967]

Figure 6: Average q-value for different frequencies of updating the target model

1.10 Conclusion

The problem is considered solved after 602 episodes as shown in Fig7. The final hyperparameters chosen were learning rate equal to 0.001, discount factor of 0.99, memory size 2000 and target update frequency equal to 1. Figure 8 depicts the q-values and scores over episodes for the best combination of hyperparameters for two different values of memory, 1000 and 2000.

```
episode: 690   score: 200.0  q_value: [138.18511988]   memory length: 2000
episode: 691   score: 200.0  q_value: [138.35355207]   memory length: 2000
episode: 692   score: 200.0  q_value: [138.51985611]   memory length: 2000
episode: 693   score: 200.0  q_value: [135.94558985]   memory length: 2000
episode: 694   score: 200.0  q_value: [133.50238778]   memory length: 2000
episode: 695   score: 200.0  q_value: [132.0648555]   memory length: 2000
episode: 696   score: 200.0  q_value: [131.3817135]   memory length: 2000
episode: 697   score: 200.0  q_value: [131.13095607]   memory length: 2000
episode: 698   score: 200.0  q_value: [130.97847487]   memory length: 2000
episode: 699   score: 200.0  q_value: [129.55141614]   memory length: 2000
episode: 700   score: 116.0  q_value: [127.95805352]   memory length: 2000
episode: 701   score: 200.0  q_value: [127.95610384]   memory length: 2000
episode: 702   score: 200.0  q_value: [128.4601735]   memory length: 2000
solved after 602 episodes
```

Figure 7: Problem is solved after 602 episodes

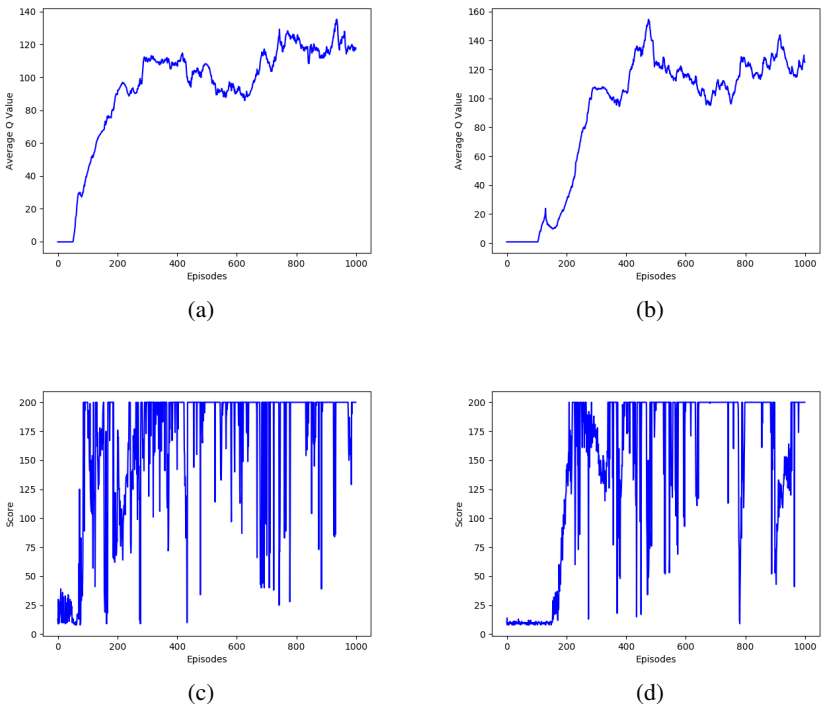

Figure 8: Left side: q values (up) and scores (down) over episodes for the best model and combination of hyperparameters so far. Right side: q values (up) and scores (down) over episodes for the best model and combination of hyperparameters so far by increasing the memory to 2000.