University of Freiburg

Deep Learning Lab

Neurorobotics/Robotics

Assignment 4

Author:
Mostafa Hussein
Eduardo Alvarado

Supervisor:
Andreas Eitel

January 22, 2018

Abstract

This report is summarizing the task of Assignment 4 which was implementing a Deep Q Network using $tensorflow$ and $keras$ package for training a q-learning agent to solve a simple maze task. The report is divided into 4 sections as follows: Section 1 is the solution of the computational task of the exercise sheet, section 2 represents a simple explanation of the implementation, section 3 represents the designed CNN and the implementation, and section 4 represents the results and the evaluation of the performance.

1 Q-Learning Task

Grid-World Environment:
All transitions have immediate reward of -1, only transitions within the goal-state are free (reward 0), discounting factor $\gamma = 0.5$.

1.1

Update Rule of Q-Learning:
$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha(r + \gamma max_{a'} Q_t(s^{'}, a^{'}) - Q_t(s,a))$
By using the given annotations, where denoted as the transition from a state i to a state j using action u and observing immediate reward r(i, u)
$Q_{t+1}(i,u) \leftarrow Q_t(i,u) + \alpha(r + \gamma max_{u'} Q_t(j, u^{'}) - Q_t(i,u))$

The transitions to and within the goal state can be handled by the Q-value. Simply, by intializing our q-values, all of the (state, action) pairs have a value of 0. Among training the agent, these q-values are updated except the goal state. When the terminal (goal) state is reached, the algorithm starts another new episode from a random initial state and then it keeps repeating until the values converges or there is no more change in the policy or the number of training episodes is reached. This means, by the end of the training, the goal state will have a 0 reward, which will be higher than all other states inside the grid, thus, the agen will always seek reaching to the goal state, and when it is there, it will never go out from it, because it has the highest reward between the neighbouring states.

1.2

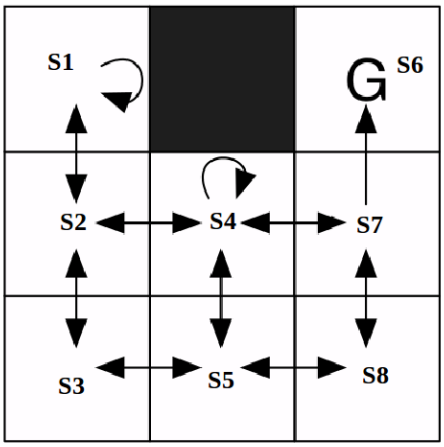


Figure 1: Grid-world

Starting with a zero-initialized Q-function, the agent starts in the upper left corner, moves a cell down (S1→S2), one cell to the right(S2→S4), tries to move upwards, fails and ends in the same cell(S4→S4), moves a cell right (S4→S7) and finally moves a cell upwards (S7→S6) into the goal state, ending this episode.

	up ↑	**right** →	**left** ←	**down** ↓
s1	0	0	0	0
s2	0	0	0	0
s3	0	0	0	0
s4	0	0	0	0
s5	0	0	0	0
s6	0	0	0	0
s7	0	0	0	0
s8	0	0	0	0

Table 1: State-Action Q-Values

Initial State: S1, action (down) S1 → S2
$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha(r + \gamma max_{a'} Q_t(s^{'}, a^{'}) - Q_t(s,a))$
$Q_{t+1}(s1, down) \leftarrow Q_t(s1, down) + \alpha(r + \gamma max_{a'}(Q_t(s2, up), Q_t(s2, down), Q_t(s2, right)) - Q_t(s1, down))$
$Q_{t+1}(s1, down) \leftarrow 0 + 1(-1 + 0.5 max_{a'}(0,0,0) - 0)$
$Q_{t+1}(s1, down) = -1$

Current State: S2, action (right) S2 → S4
$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha(r + \gamma max_{a'} Q_t(s^{'}, a^{'}) - Q_t(s,a))$
$Q_{t+1}(s2, right) \leftarrow Q_t(s2, right) + \alpha(r + \gamma max_{a'}(Q_t(s4, up), Q_t(s4, down), Q_t(s4, right), Q_t(s4, left)) - Q_t(s2, right))$
$Q_{t+1}(s2, right) \leftarrow 0 + 1(-1 + 0.5 max_{a'}(0,0,0,0) - 0)$
$Q_{t+1}(s2, right) = -1$

Current State: S4, action (up) S4 → S4
$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha(r + \gamma max_{a'} Q_t(s^{'}, a^{'}) - Q_t(s,a))$
$Q_{t+1}(s4, up) \leftarrow Q_t(s4, up) + \alpha(r + \gamma max_{a'}(Q_t(s4, up), Q_t(s4, down), Q_t(s4, right), Q_t(s4, left)) - Q_t(s4, up))$
$Q_{t+1}(s4, up) \leftarrow 0 + 1(-1 + 0.5 max_{a'}(0,0,0,0) - 0)$
$Q_{t+1}(s4, up) = -1$

Current State: S4, action (right) S4 → S7
$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha(r + \gamma max_{a'} Q_t(s^{'}, a^{'}) - Q_t(s,a))$
$Q_{t+1}(s4, right) \leftarrow Q_t(s4, right) + \alpha(r + \gamma max_{a'}(Q_t(s7, up), Q_t(s7, down), Q_t(s7, left)) - Q_t(s4, right))$
$Q_{t+1}(s4, right) \leftarrow 0 + 1(-1 + 0.5 max_{a'}(0,0,0) - 0)$
$Q_{t+1}(s4, right) = -1$

Current State: S7, action (up) S7 → S6
$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha(r + \gamma max_{a'} Q_t(s^{'}, a^{'}) - Q_t(s,a))$
$Q_{t+1}(s7, up) \leftarrow Q_t(s7, up) + \alpha(r - Q_t(s7, up))$
$Q_{t+1}(s7, up) \leftarrow 0 + 1(0 - 0)$
$Q_{t+1}(s7, up) = 0$

	up ↑	**right →**	**left ←**	**down ↓**
s1	0	0	0	**-1**
s2	0	**-1**	0	0
s3	0	0	0	0
s4	**-1**	**-1**	0	0
s5	0	0	0	0
s6	0	0	0	0
s7	**0**	0	0	0
s8	0	0	0	0

Table 2: Updated State-Action Q-Values

2 Convolutional Neural Network

This exercise was to implement an agent using a DQN to solve a simple maze task. Firstly, there is no dataset, however, the agent has to react through exploring the space and getting rewards, and this is the main idea of Deep Q-Learning. A *Simulator* and a $TransitionTable$ were initialized at the beginning, and we begin by initializing the model using *keras* package, which is an API that acts over $TensorFlow$. *Dense*, *Activation*, $Conv2D$, $MaxPooling2D$, *Dropout* and *Flatten* layers were imported from $keras.layers$. The agent is using ϵ-greedy algorithm to take its actions. The main idea is to balance between exploration and exploitation in the environment, and this will be explained later in the implementation section.

3 Design and Implementation

3.1 Design

The designed DQN consists of three same padded convolutional layers with number of features 32, 64, 64 and $kernel\_size$ 4x4, 4x4, 3x3, and strides of 2, 2, and 1 respictevly, each followed by a $ReLu$ activation function. Afterwards, we added a $Flatten$ layer, followed by a *Dense* layer of number of units equals to number of possible actions, which will give out an output of predicted Q-values. Then the best action will be chosen according to the highest Q-value. The layer parameters were randomly tried out from a range of values, and these parameters we have, were the ones who gave out best results and

performance. In addition, we used *Adam* optimizer with a *learning_rate* of $1e-6$ and a MSE loss. The model was trained with $steps = 1,000,000$ and a $batch\_size = 32$.

3.2 Implementation

3.2.1 Decaying ϵ-greedy Algorithm

The main concept of using this algorithm was to balance between exploration and exploitation of the environment. The agent needs to explore the environment and try different (states, action) pairs, to train itself towards the goal. This is achieved by an epsilon-greedy method for exploration during training. This means that when an action is selected in training, it is either chosen as the action with the highest q-value, or a random action. Choosing between these two is probablistic and based on the value of epsilon, and epsilon is annealed during training such that initially, lots of random actions are taken (exploration), but as training progresses, lots of actions with the maximum q-values are taken (exploitation). In many cases the ϵ is kept as a fixed number in range 0 and 1, And usually, the exploration diminishes over time, so that the policy used asymptotically becomes greedy and therefore ($Qk \rightarrow Q*$) becomes optimal. This can be achieved by making ϵk approach 0 as k grows, and for sure it is all dependent on the number of iterations. In this exercise, ϵ was initialized by a starting value of 1, and an end value of 0.1. Among the training, it is annealed by a decaying factor of 0.999982 for $130,000$ steps until it reaches a constant end value of 0.1. This will ensure that the agent will depend firstly on random actions to have a better exploration on the environment, which will help in training itself, and after $50,000$ steps, it will depend more on actions which are predicted from the model itself, and this is the exploitation part. Below is a graph showing the decaying of the epsilon among the number of steps.

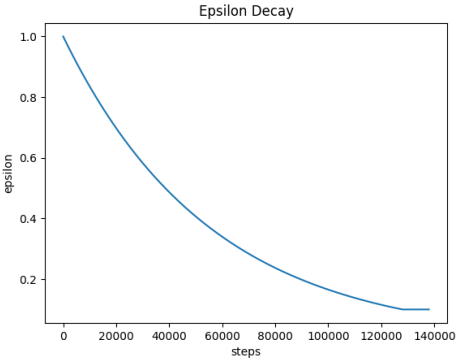


Figure 2: Epsilon Decay

3.2.2 Target Network

During training of our algorithm we set targets for gradient descend as:

$$Q(s,a) \rightarrow r + \gamma max_a Q(s',a)$$

In TD error calculation, target function is changed frequently with DNN. We see that the target depends on the current network. A neural network works as a whole, and so each update to a point in the Q function also influences whole area around that point. And the points of Q(s, a) and Q(s', a) are very close together, because each sample describes a transition from s to s'. The issue is that at every step of training, the Q-network's values shift, and if we are using a constantly shifting set of values to adjust our network values, then the value estimations can easily spiral out of control. The network can become destabilized by falling into feedback loops between the target and estimated Q-values. This leads to a problem that with each update, the target is likely to shift. The network sets itself its targets and follows them, and this can lead to instabilities, oscillations or divergence. Unstable target function makes training difficult. To overcome this problem, researches proposed to use a separate target network for setting the targets. This network is a mere copy of the Q network, but frozen in time. It provides stable $\tilde{Q}$ values and allows the algorithm to converge to the specified target:

$$Q(s,a) \rightarrow r + \gamma max_a \tilde{Q}(s',a)$$

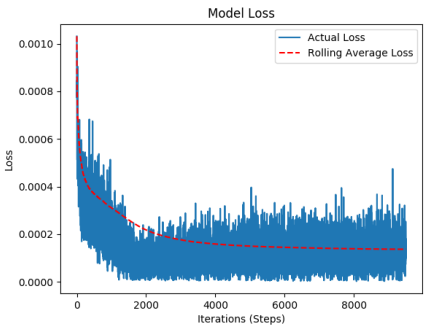
Accordingly, we used a separate network to estimate the TD target. This target network has the same architecture as the function approximator but

with frozen parameters. Target Network technique fixes parameters of target function and replaces them with the latest network every ten thousands steps. In order to mitigate that risk, the target network's weights are fixed, and only periodically or slowly updated to the primary Q-networks values. Every T steps (a hyperparameter) the parameters from the Q network are copied to the target network. In this way, training can proceed in a more stable manner because it keeps the target function fixed (for a while).

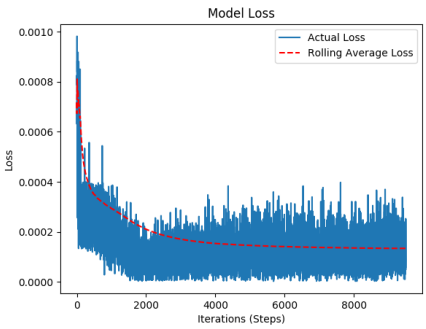
3.2.3 Calculating the loss

An step in the environment gives out the following four elements: state, action, reward and next state. The neural network has as many output units as actions possible. This makes possible to find the best action with one forward pass through the network. The loss is calculated by calculating the difference for that action and for the others it is zero. But in Keras one should define training sample and target sample pairs. Therefore the target is calculated in two steps:

1. Forward pass through the network and calculate action-values for each action.
2. Modify the action-value which corresponds to the actual observation by applying the update rule of the Q-values.



(a) Loss - without Target Network



(b) Loss - with Target Network

Figure 3: Model Loss - with and without Target Networks

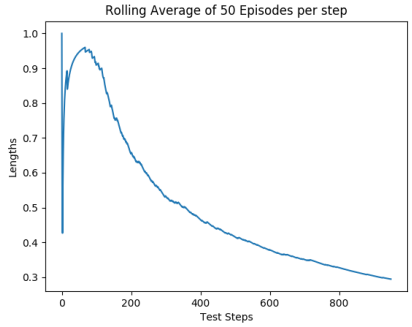
3.2.4 Parameters

- learning rate: 0.001
- target network update frequency: 10,000
- number of iterations: 1,000,000
- epsilon value at the very beginning (in epsilon-greedy): 1.0 (initial exploration)
- smallest epsilon value: 0.1 (final exploration)
- epsilon annealing value: 0.999982 (decay factor)
- the number of steps to achieve the smallest epsilon: 130,000 (final exploration frame)
- gamma: 0.99 (discount factor)
- evaluation frequency to measure the current performance: 1,000 steps
- number of episodes in one evaluation: 50
- Optimizer: Adam in Keras

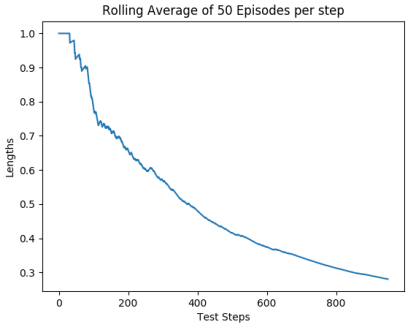
4 Results

4.1 Training

The agent was trained twice, one with just the Q-model, and the other one was with adding the target network. Every 1000 steps, the agent was evaluated to see its performance during training, and the metrics were logged. For every test step, the agent was evaluated on 50 complete episodes, and the average of the achieved rewards as well as the number of steps it takes to reach a terminal state, or an early stop of 75 was plotted as shown below in the figures. It is clearly obvious from the below plots, that using a target network is more stable for the training and it achieves better results. As shown in Figure 4, we can observe that using the target network, is leading to a convergence from around nearly 650,000 steps, however, around 800,000 steps with only using the Q-model, additionally, the average number of steps for reaching the terminal state is a little bit higher with using the Q-model only, this is because the agent is not taking optimal path towards the goal. Moreover, the average of the rewards is illustrated also in Figure **??**, and it is showing the same.

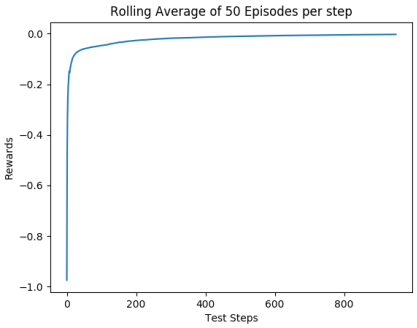


(a) Average Episode Lengths - without Target Network

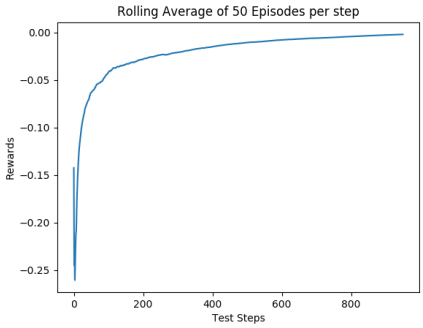


(b) Average Episode Lengths - with Target Network

Figure 4: Average Episode Lengths - with and without Target Networks



(a) Average Episode Rewards - without Target Network

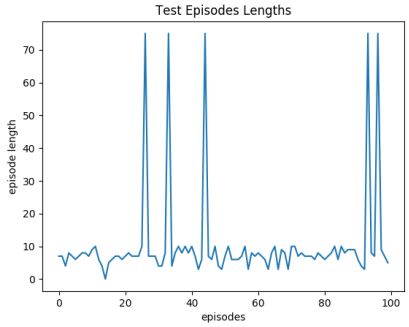


(b) Average Episode Rewards - with Target Network

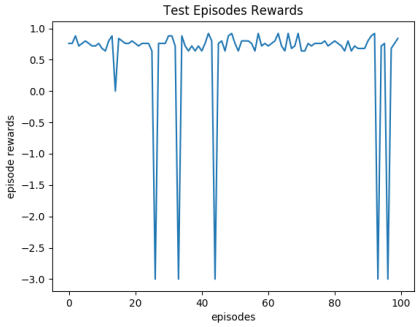
Figure 5: Average Episode Rewards - with and without Target Networks

4.2 Testing

Finally, the agent was tested with our trained models (with and without Target Network) for 100 episodes, and the episodes' length, reward were logged and illustrated in the figures below. In Figure 6, we can see an illustration which is showing how many episodes reach a terminal state, and how many failed to reach in a comparison of testing with and without target network.

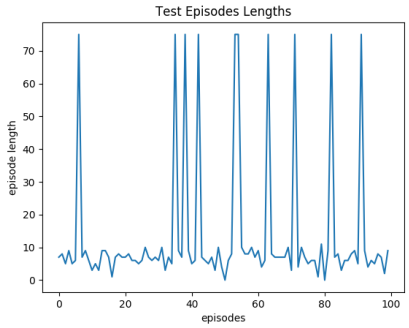


(a) Episodes Lengths

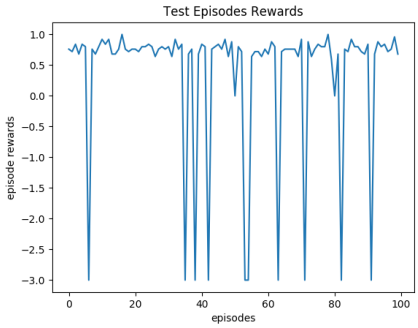


(b) Episodes Rewards

Figure 6: Testing with Target Network



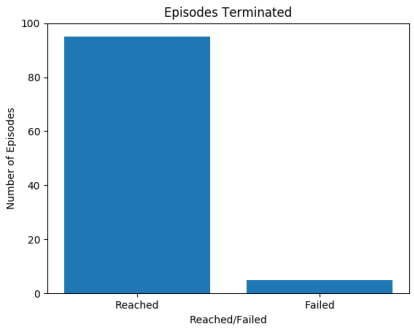
(a) Episodes Lengths



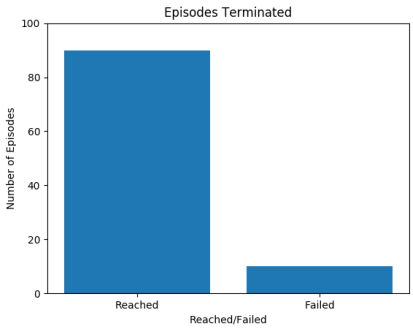
(b) Episodes Rewards

Figure 7: Testing without Target Network

It is clearly shown from the above graphs, that the testing with the model which included a target network gave better results than the other one. In the below figure 8, is a bar plot which is showing the number of completed episodes during the testing for both models.



(a) with Target Network



(b) without Target Network

Figure 8: Number of completed episodes