

Reinforcement Learning using Deep Q - Learning

Saniea Akhtar, Hendrik Vloet

January 23, 2018

1 Introduction

1.1 Short Recap of Q - Learning

Q - Learning is one of the basic model free reinforcement learning techniques. Reinforcement learning is different from supervised in a number of ways the major difference being in the data that is provided to the system. In supervised learning, correct input - output pairs of data are provided to the model in order to help it to learn correct classification of data. In reinforcement learning, no pairs of data are provided, nor are the actions of the model directly corrected. Rather a model is created that is provided with a reward signal and a series of sequential, non-i.i.d data, collected during execution. The focus of the model is mostly on finding a balance between learning (collection of data) and using the already collected knowledge to reach the required goal (i.e. Exploration/Exploitation-Tradeoff).

Most reinforcement learning tasks are formulated as Markov decision process (MDP) problem. The MDPs provide a mathematical framework for modeling a decision making situations where outcomes are partly random and partly under the control of a decision maker. The model follows the underlying Markov property “The future is independent of the past given the present”.

An MDP framework has five basic elements:

- \mathcal{S} : the states of the system
- \mathcal{A} : the actions that can be taken by the agent
- \mathcal{P} : the transition probability: $\mathcal{P}_{ss'}^a = \mathbb{P} [S_{t+1} = s' \mid S_t = s, A_t = a]$
- \mathcal{R} : the reward function: $\mathcal{R}_s^a = \mathbb{E} [R_{t+1} \mid S_t = s, A_t = a]$
- γ : the discount factor: $\gamma \in [0,1]$

Generally, the following MDP is formulated for a Q - learning problem:

$$Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma(\max_{A'} Q(S', A') - Q(S, A)))$$

This is also the policy the agent uses to select actions given the state that its currently in. Using such a function, the optimal policy can be determined by simply selecting the action with the highest value in each state. This structure is able to determine the expected rewards of all the available actions without requiring a model of the system. Q - learning can also stochastic transition problems without any changes to the structure. For any finite MDP, Q - learning will eventually find an optimal policy.

2 Tasks

2.1 Task 1: Q - Learning

Task 1.1 Write down the update-rule of Q-Learning for updating the Q-function after a transition from a state i to a state j using action u and observing immediate reward $r(i; u)$. How would you handle transitions to or within the goal state (which is absorbing, i.e. the agent can never transition out of it)?

The following update rule will be used for the given model:

$$Q(i, u) \leftarrow Q(i, u) + \alpha(r(i, u) + \gamma(\max_{u'} Q(j, u') - Q(i, u)))$$

Or slightly more specifically:

$$Q(i, u) \leftarrow Q(i, u) + \alpha(r(i, u) + 0.5 * (\max_{u'} Q(j, u') - Q(i, u)))$$

Using the above Q - function, transitions to the goal state will be rewarded with a high reward value, in this case of 0. During the execution of the network, while the Q - values are being updated, the value of the goal state will remain the highest (in this case, 0), while the Q - values of the other states will continue to grow smaller with each step (in this case, become negative). As a result, once the agent enters the goal state, the cost to leave it will be too high for it to consider moving out. As a result, the agent will remain in the goal state almost always. Also, the program should terminate once the goal state is reached in order to ensure correct execution.

The acting policy is also important for this particular case. For example, if the agent would pick its actions in a non-greedy fashion, e.g. using an ϵ - greedy policy, then there would still be a chance that the agent leaves the goal state according to a non - optimal choice. But usually during the execution, the agent always uses the optimal choice based on the Bellman Optimality Equation, so the probability of this happening is very low.

Task 1.2 Starting with a zero-initialized Q - function, the agent starts in the upper left corner, moves a cell down, one cell to the right, tries to move upwards, fails and ends in the same cell, moves a cell right and finally moves a cell upwards into the goal state, ending this episode. Determine, which Q-values would have been changed during this episode when using Q - learning with a learning rate of 1.0. Specify the improved Q - function after this initial episode.

At the beginning of this episode, the matrix holding Q - values would consist entirely of zeros, i.e. :

$$Q = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

With each step the agent takes, the values of the respective cells, in the format Q(row, column), change as follows :

$$\begin{aligned} Q(1, 1) &= 0 + 1 \cdot (-1 + 0.5 \cdot 0 - 0) = -1 \\ Q(2, 1) &= 0 + 1 \cdot (-1 + 0.5 \cdot 0 - 0) = -1 \\ Q(2, 2) &= 0 + 1 \cdot (-1 + 0.5 \cdot 0 - 0) = -1 \\ Q(2, 2) &= -1 + 1 \cdot (-1 + 0.5 \cdot 0 - (-1)) = -1 \\ Q(2, 3) &= 0 + 1 \cdot (-1 + 0.5 \cdot 0 - 0) = -1 \\ Q(1, 3) &= 0 + 1 \cdot (-1 + 0.5 \cdot 0 - 0) = -1 \end{aligned}$$

As shown in the above equations, the Q - values change with each step the agent takes. The value for (2,2) changes twice since the agent attempts to move into a cell that is blocked. Once the agent reaches the goal state, the episode ends. The final matrix will hold the following values :

$$Q = \begin{bmatrix} -1 & 0 & 0 \\ -1 & -1 & -1 \\ 0 & 0 & 0 \end{bmatrix}$$

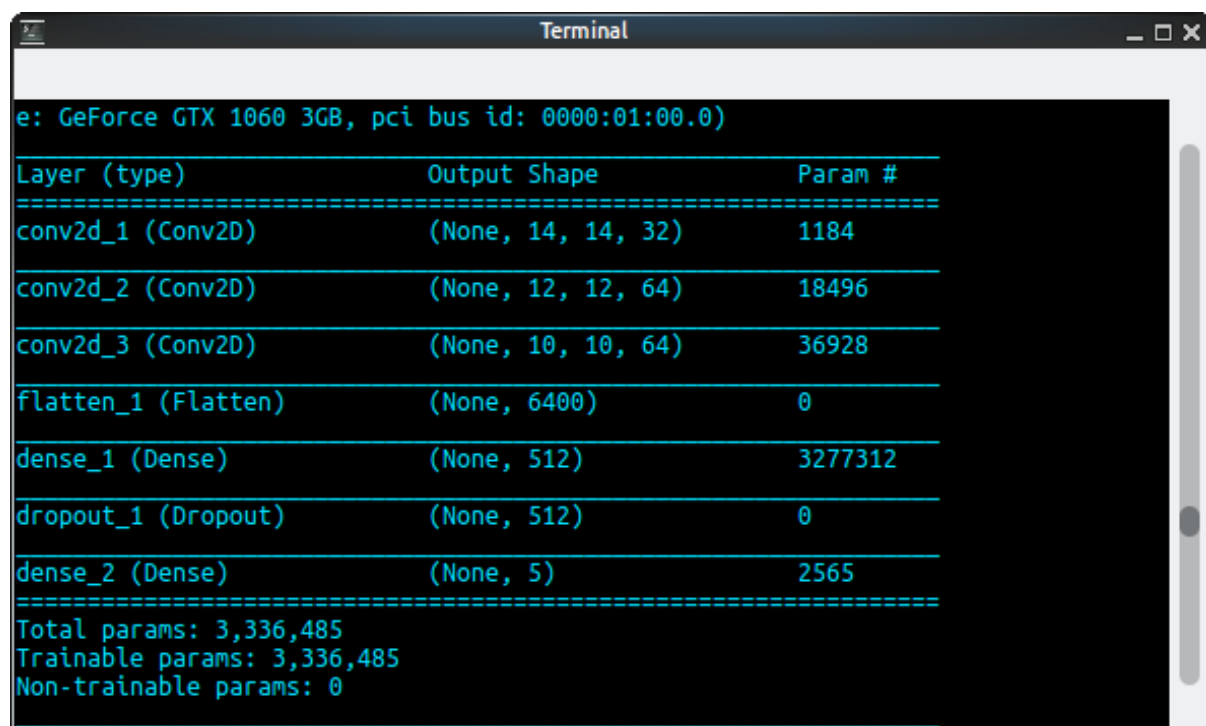
2.2 Task 2: Implementing Deep Q - Learning

This task required the implementation of a Deep Q - Learning network in the given framework. In order to evaluate the performance of the agent, we compared it with the performance of the A*-Solver of the previous task. Overall we could observe, that a well trained DQN Agent could perform with roughly the same efficiency as the optimal acting A*-Solver. One remark regarding the comparison charts: the A*-Solver runs before the agent, so their setup is slightly different. The charts are supposed to give an intuition about how the agent approximates the performance of an A*-Solver. We also made a slight modification to the target reward(we changed it from 1.0 to 10.0). Also interesting to see was, how noisy the reward was.

Overall we achieved good, depending strongly on the invested time of training.

Note: All models are in their respective folders in the *working_models* directory

2.2.1 Used Architecture:



The image shows a terminal window with a black background and cyan text. At the top, it displays GPU information: 'e: GeForce GTX 1060 3GB, pci bus id: 0000:01:00.0'. Below this is a table with three columns: 'Layer (type)', 'Output Shape', and 'Param #'. The table lists seven layers: conv2d_1 (Conv2D), conv2d_2 (Conv2D), conv2d_3 (Conv2D), flatten_1 (Flatten), dense_1 (Dense), dropout_1 (Dropout), and dense_2 (Dense). At the bottom, it summarizes the total, trainable, and non-trainable parameters.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 14, 14, 32)	1184
conv2d_2 (Conv2D)	(None, 12, 12, 64)	18496
conv2d_3 (Conv2D)	(None, 10, 10, 64)	36928
flatten_1 (Flatten)	(None, 6400)	0
dense_1 (Dense)	(None, 512)	3277312
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 5)	2565

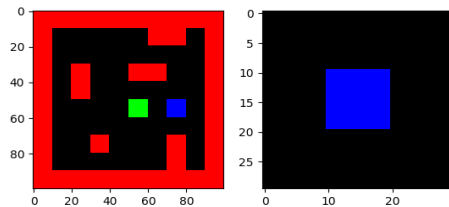
=====
Total params: 3,336,485
Trainable params: 3,336,485
Non-trainable params: 0

2.2.2 Map A comparison between linear and exponential ϵ -decay

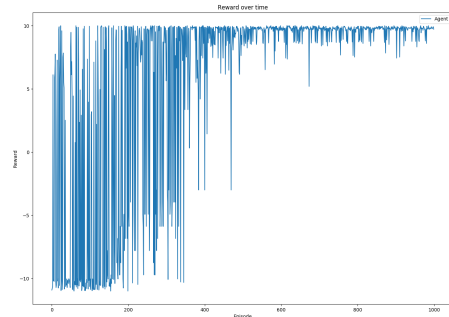
Map A

- 1000 Training episodes
- linear decaying ϵ -greedy

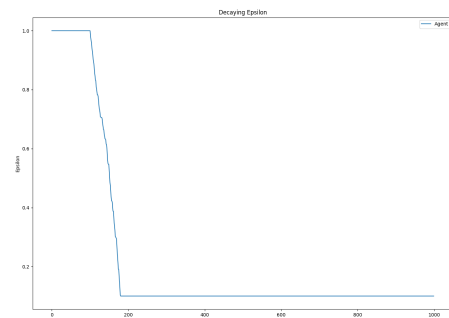
Map Layout



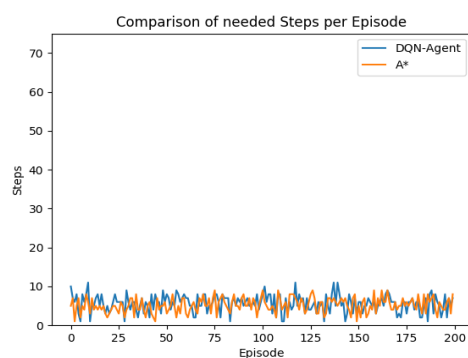
Reward over time



Decaying Epsilon per episode (y-label on picture is wrong)



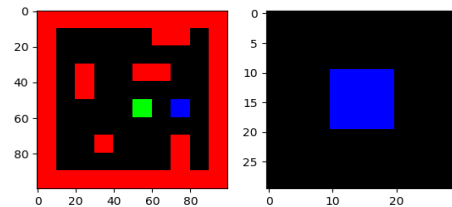
Comparison of the steps needed



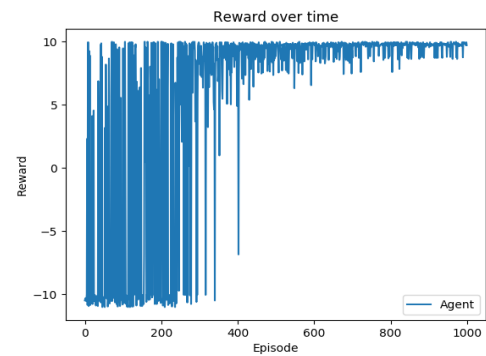
Map A

- 1000 Training episodes
- exponential decaying ϵ -greedy

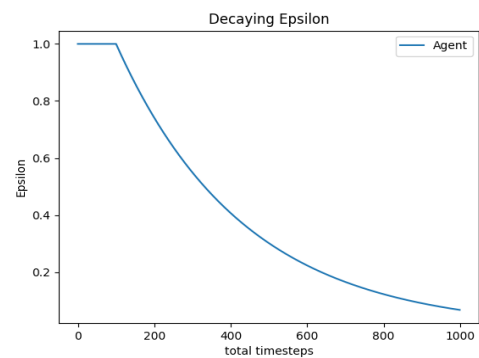
Map Layout



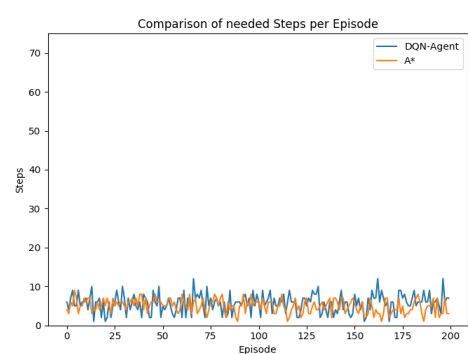
Reward over time



Decaying Epsilon per episode (y-label on picture is wrong)



Comparison of the steps needed



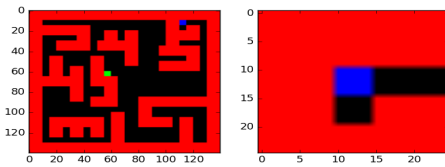
2.2.3 Map B comparison between linear and exponential ϵ -decay

When we trained the first time with Map B and linearly decreasing epsilon, we saw that our initial chosen length of episodes wouldn't be enough to train on in order to achieve good results. Because of that we changed the amount of episodes from 1000 to 4000 and started the training for Map B. Unfortunately there was not enough time to retrain Map A with 4000 episodes as well, but in the end it shows how important the length of training can be.

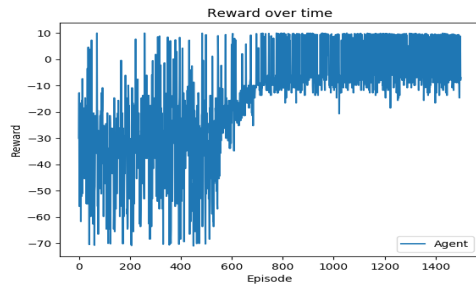
Map B

- 1500 Training episodes
- linear decaying ϵ -greedy

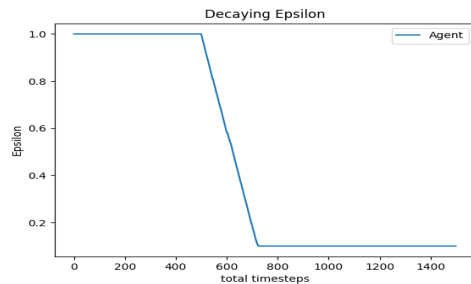
Map Layout



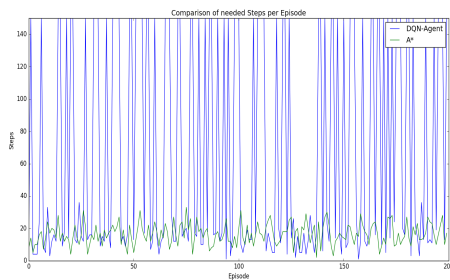
Reward over time



Decaying Epsilon per episode (y-label on picture is wrong)



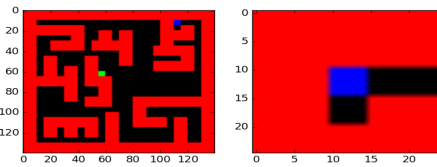
Comparison of the steps needed



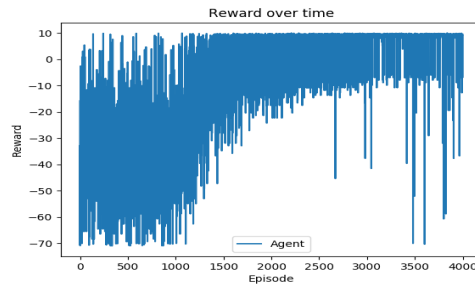
Map B

- 4000 Training episodes
- exponential decaying ϵ -greedy

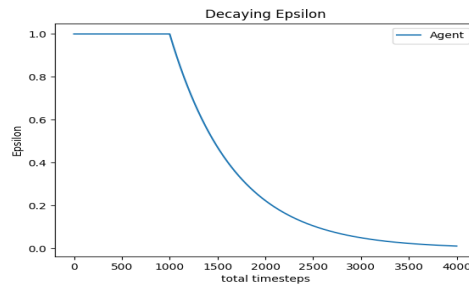
Map Layout



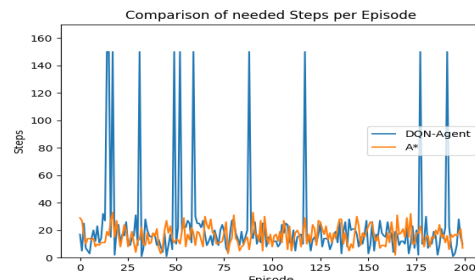
Reward over time



Decaying Epsilon per episode (y-label on picture is wrong)



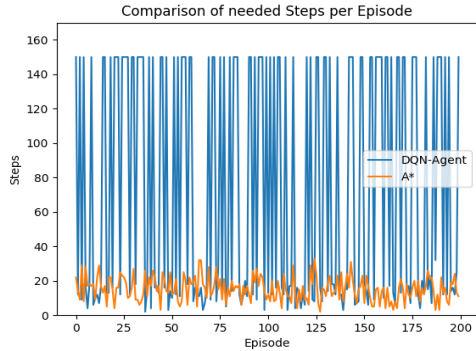
Comparison of the steps needed



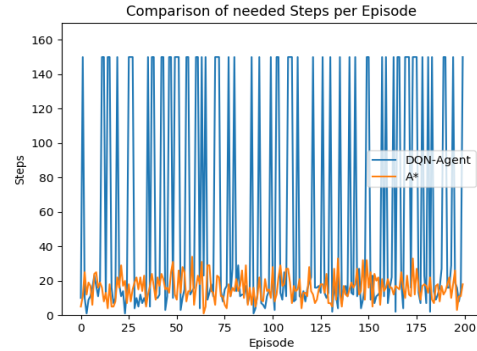
2.2.4 Training Demonstration

In this section we will provide some charts about the learning progress of the agent with exponential decaying epsilon while it trains on map "B". During training, the model is saved every N Episodes (in our case 50). And in order to monitor it's progress, we evaluated the agent from time to time. Evaluation before 2000 episodes of training, the agent performed below 50% success rate and was considered by us not being of interest. Every time the agent hits an amount of 150 steps in one episode or accumulates a total reward smaller than -70, the training episode is stopped.

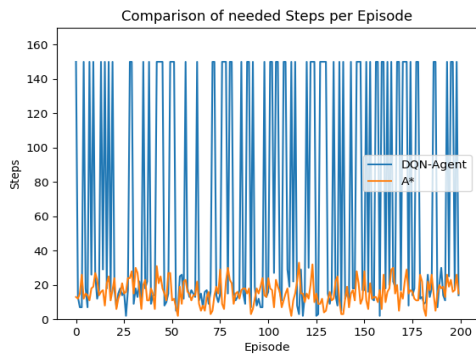
Evaluating after 2000 steps of training



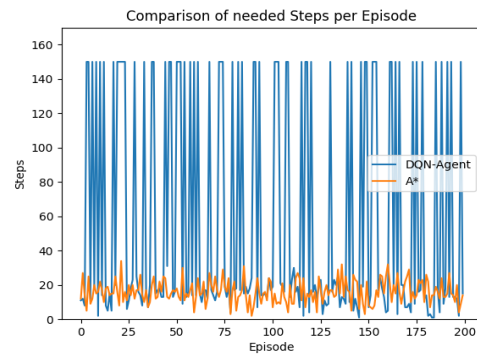
Evaluating after 2400 steps of training



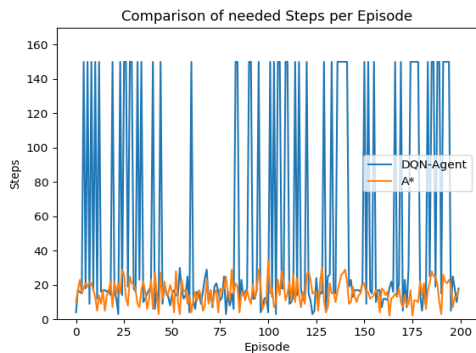
Evaluating after 2100 steps of training



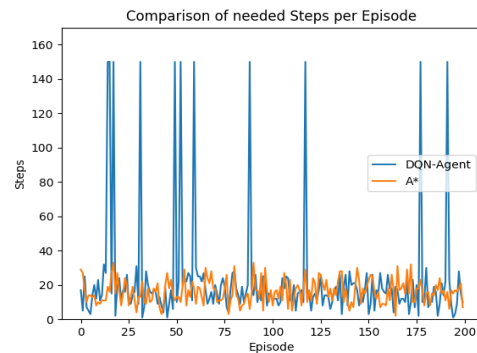
Evaluating after 2500 steps of training



Evaluating after 2300 steps of training



Evaluating after 4000 steps of training



2.2.5 Further Examination

We tried again to change the map after the training and also changed the target position, but as expected, the agent fails due to its now unknown environment which he did no train upon.

For a changed map, the agent could not manage to reach the goal state, because it did not train on the new map.

For a changed target, the agent simply fails to see it because the agent expects the target to be at the old position

