Project report

Exercise 4
Deep Q Learning

Deep Learning Lab Course 2017
University of Freiburg

Autors: Fabien Jenne Johannes Engler
3725777

Robotics & Neurorobotics track

22.01.18

Contents

1. Outline

This exercise addresses a reinforcement learning technique called Q-learning.
The first task is about the manual calculation of Q values for updating a Q-table for a given environment. Task 2 approaches Deep Q Learning, which is a deep learning implementation of Q-learning using neural networks.

2. Task 1

The solutions for task 1 can be found in the corresponding file `Sheet 4 Exercise 1.pdf` in the git-repository.

3. Task 2

The code for this exercise can be found under:
`https://github.com/johannes5117/exercise4robotics`

The task of this exercise is to implement a deep neural network which implements Q-learning. As the state space becomes larger, the dimensions of the Q-table in traditional Q-learning grows to infinity, making it infeasible to use.
The idea behind deep Q learning is to represent the Q table as a neural network and let the network predict the value of a Q-table entry $Q(s, a)$ for a certain action a given state s.
This is done by implementing the Q-function:

$$Q(s,a)' \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot (\text{Reward}(s') + \gamma \max_{a'} Q(s',a'))$$

for random batches of states s and taken actions a while leaving the other values unchanged:

$$Q(s,a) \leftarrow Q(s,a).$$

Performing this assignment iteratively and using the difference:

$$Q(s,a)' - Q(s,a) = \alpha \cdot (\text{Reward}(s') + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

as training loss, the network should more and more resemble the true Q-table.

4. Implementation

We implement the NN in Python using the Keras framework which is based on the Tensorflow API.

Our network has the following final architecture:

- Fully connected layer: 128 filters, ReLu activation
- Fully connected layer: 256 filters, ReLu activation
- Fully connected layer: 256 filters, ReLu activation
- Loss layer: Mean squared error loss, Softmax activation

The `Adadelta` optimizer is used. The training is done for 1000 episodes, each evolving until the goal state is reached or the maximum number of steps is reached.

Unfortunately, we didn't manage to get the Q-Net to predict the optimal next action as expected. The code runs though and performs the training. We tried really hard for several days of work to get it running, but is just doesn't do what we want it to. So here, help is greatly appreciated, we couldn't find the fault on our own.

We tried batch processing as well as going over each batch individually ($\rightarrow$ see code). Furthermore, we altered the hyperparameters α, γ, ε, ε_{min} $\varepsilon_{\text{decay}}$. Additionally we changed the complete network architecture, trying convolutions, as well as solely fully-connected layers. Moreover, we also modified the number of steps, episodes, as well as the max-step threshold.

5. Evaluation

The plot in figure 0.1 shows the evolution of the loss during training. As can be seen in the figure, the loss decreases steadily, which means that the Qnet more and more resembles the Q-function. However, the number of steps needed to solve the task don't correspond to this observation, as can be seen in figure 0.2. In fact, the performance is best during the first 100 episodes, where epsilon is still close to 1.0. (see figure 0.3) But that actually means taking random actions.
We still don't know why our implementation doesn't work properly, and would appreciate your feedback and ideas a lot.

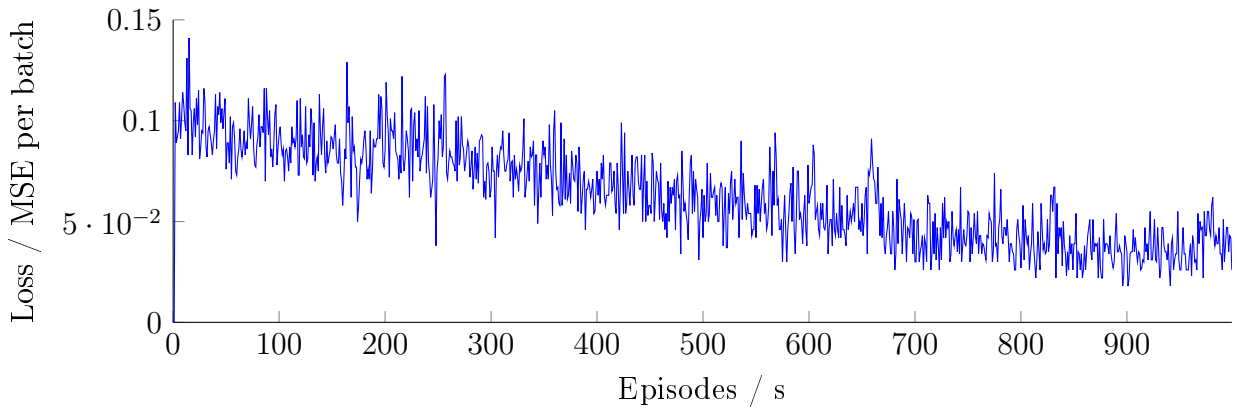

Figure 0.1: Training loss of the DQN.

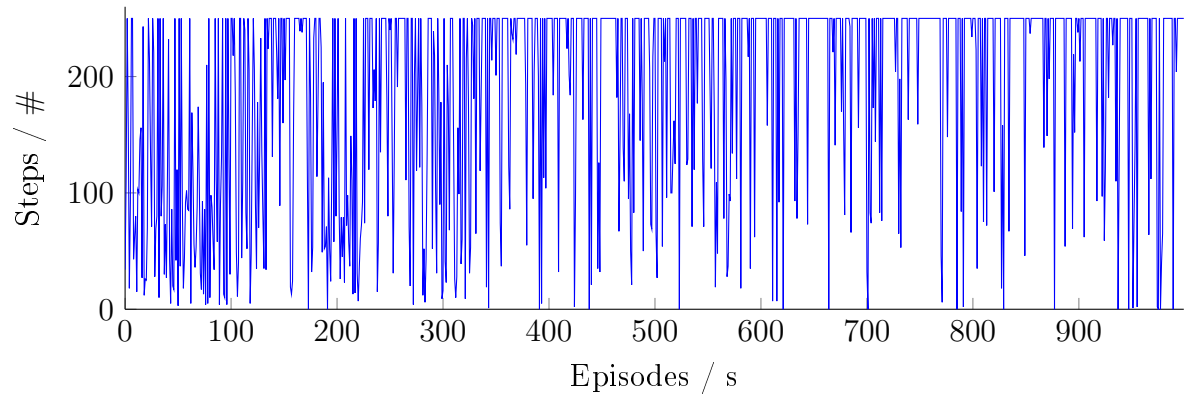

Figure 0.2: Number of steps until goal is reached in each episode.
Max-step is set to 250.

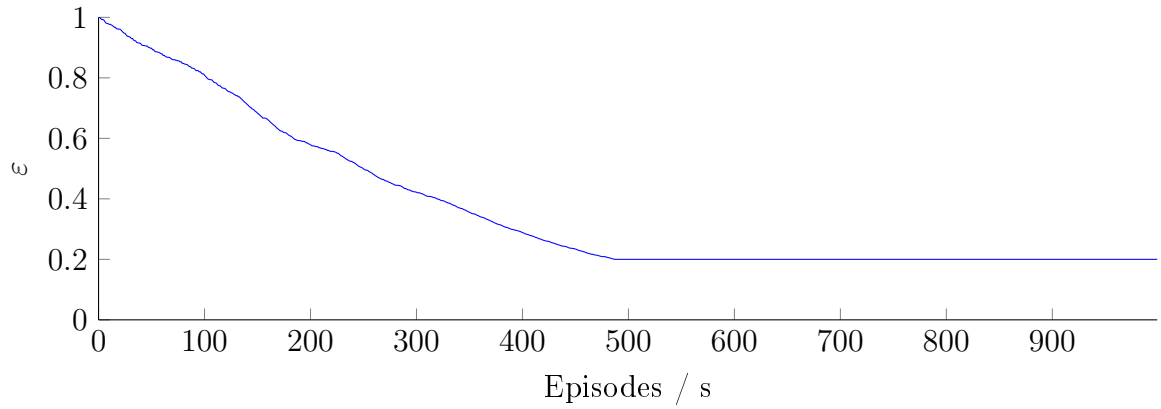

Figure 0.3: Evolvement of ε during the training.