DEEP LEARNING LAB COURSE

Assignment 4 - Q-Learning

Daniel Bunzendahl, Max Schlichting

January 21, 2018

1 INTRODUCTION

In this fourth assignment, our task was to implement a deep q-learning agent for the same maze task as in assignment three. Again, we were provided with the simulation framework. The agent gets to see a local view of the map (plus some history) and has to predict the optimal action. The goal of this exercise was, to train the agent without pre-generated training data, i.e. the agent has to learn the optimal action by itself (without the A^* planner).

2 BACKGROUND

In order to choose an action, the agent learns a q-value for each state and action. The optimal action a^* is considered the action which maximizes the q-value in the current state. Hence in state s the agent performs an action a^* according to

$$a^* = \arg\max_{\text{a}} Q(s,a)$$

The update-rule of q-learning, i.e how to update the q-function after a transition from a state s to a state s' using action a and observing immediate reward $r(s,a)$ is given by

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

Here we call $\alpha \in [0,1]$ learning rate and $\gamma \in [0,1]$ discount factor. Transitions within the goal state can be updated the same way. However, as the goal state is absorbing, we have $s' = s$.

The assignment also asked for the updated q-value of the small grid-world depicted in Figure 1. Initially all q-values are set to zero. The learning rate α is 1.0 and the discounting factor γ is 0.5. The agent moves from the upper left corner s_{11} down to s_{21}, to the right to s_{22}, tries to go up but again ends in s_{22} and finally goes right to s_{23} and up to s_{13}. Each action gives

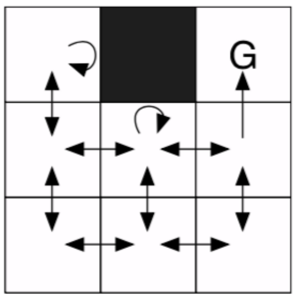

Figure 2.1: The simple gridworld.

a reward of -1, only actions within the goal state (not to the goal state) give a reward of 0. According to the update-rule this gives us:

$$\begin{aligned}
Q(s_{11}, \downarrow) &= -1 + 0.5 \cdot 0 = -1 \\
Q(s_{21}, \rightarrow) &= -1 + 0.5 \cdot 0 = -1 \\
Q(s_{22}, \uparrow) &= -1 + 0.5 \cdot 0 = -1 \\
Q(s_{22}, \rightarrow) &= -1 + 0.5 \cdot 0 = -1 \\
Q(s_{23}, \uparrow) &= -1 + 0.5 \cdot 0 = -1
\end{aligned}$$

All other q-values remain unchanged (zero).

3 Implementation

We again separated the training and testing of the agent into two different files. The file *train_agent.py* trains our agent according to the parameter setting in *utils.py*. The file *test_agent.py* runs 300 episodes and computes the percentage of solved episodes. We consider an episode as solved if the agent finds the goal with less than 75 steps, performing only greedy actions with respect to the learned q-function. In the original DQN-paper, even for testing the agent did perform epsilon-greedy actions. However, we chose to perform only greedy actions to make a stronger statement of our learned q-function.

The file *agent.py* implements our agent class. This class stores the agents neural network as well as its seen history. We chose to adapt the neural network from the previous assignment: First we have two convolutional layer (eight 3×3 filters), each followed by a max-pooling layer. These are followed by two fully connected layer. All layers but the last one have a ReLU activation function. However our model did not converge and the agent did not learn to solve the maze.
With extensive help from the tutors we made the following changes: Most importantly we did not reuse the weights from computing the q-value for the current state for computing the q-values for the next state. Hence, we never updated the weights for computing *q_next*. Furthermore we had to remove the max-pooling layers and initialize all weights with a normal distribution with zero mean and standard deviation of 0.01 (the default standard deviation is

0.1). Finally, the model did only converge with a dropout layer after the first fully connected layer.

4 Evaluation

First we used an Adam-optimizer with a learning rate of 0.00001, batch size of 128 and a history-length of four. The agent performed an epsilon-greedy exploration with $\epsilon = 0.5$. The model is extremely sensitive to the hyperparameter setting and even minor changes can cause the model to diverge during training. We trained the agent for 3000 episodes. With this setting, we achieved an average rate of 80% solved episodes during testing. However, the agent often did not take the shortest path to the goal.

Next we changed the history-length to eight and again trained the agent for 3000 episodes. This time the agent solved 100% of the episodes but still did not take the shortest path to the goal. We uploaded the learned network weights to the repository.

Although it was not part of the assignment, we added a target network in order to make the learning process more stable and less sensitive for the hyperparameter setting. The goal of the target network is to have a second network to compute *q_next*. The weights of this second network are slowly updated to the weights of the primary q-network. This way the target is not constantly changing and the training becomes more stable.

With this target network we achieved surprisingly good results: The training procedure is less sensitive to hyperparameters and is much faster. We were able to remove the dropout layer, increase the learning rate to 0.1 and reduce the history size to four. After training the agent for only 200 episodes it was able to solve 100% of the episodes during testing and always took the shortest path to the goal. We again uploaded the learned network weights to the repository, i.e. you can run *target_network/test_agent.py*.

5 Conclusion

We were able to see that we can use q-learning to train a robot to solve a simple maze task without any expert providing training data. However we also experienced how sensitive the learning procedure is to the hyperparameter setting. Finally, we were able to improve the simple q-learning by implementing a target network.