Constantin Eulenstein

CID: 01804832

Reinforcement Learning (Second Half) Coursework

Question 1 (i) The loss curve obtained from online learning exhibits much higher variance than the one obtained from learning with experience replay buffer (ERB) and batch size of 100. ERB results in more stable training with less variance, because the agent trains on 100 randomized samples simultaneously instead of sequential updates based on only one state-action experience per transition. This update of the entire state action space at once leads to a more even distribution of training. In online learning updating subsequent states, which are highly correlated, heavily affects the entire function approximation. The agent gets better in one region, but might worsen the approximation in other regions of the state action space due to generalization. This, in turn, increases the variance of our curve, because the Q-values have a harder time to converge. ERB solves this issue.

Question 1 (ii) An agent trained with ERB is more efficient at improving the accuracy per episode. First, in Online learning the agent needs to revisit a state-action pair every time to actually train on that transition. With ERB, every step, we train 100 random samples at once, which leads to faster and more efficient learning per episode. Second, with ERB the network recalls and trains even rare occurrences that may be overwritten (through generalization) in online learning because they haven't been trained on for a while. Using ERB, the network can retrain on states without having to revisit these states again. This is way more efficient. Third, the algorithm does not have to correct wrong function approximations (as described for online learning in (i)) to the same extent.

Question 2 (i) It is more likely that the bottom right of the environment has more accurate Q-value predictions. The agent has been trained using a random policy and always starts in the bottom-left. Consequently, under a random policy and episode length of only 20 steps, it is unlikely that the agent passes the obstacle and explores the top-right very often. Therefore, the agent will have explored and trained way more on the bottom-right than on the top-right and will have predicted more accurate Q-values. This is also observable in the figure: In the bottom-right action 'up' exhibits the highest Q-values. The same holds true for some uppermost states on the right, which should actually predict action 'left'.

Question 2 (ii) Under a greedy policy, it is very unlikely that the agent would reach the goal state. The agent has been trained on predicting the immediate reward only (very myopic, $\gamma = 0$) and learned that the Q-values rise with decreasing distance to the goal. Under a greedy policy, the agent reduces the distance by simply moving upwards until it reaches the obstacle. From there, moving right or left reduces the immediate reward and, therefore, the Q-value for action 'up' has the highest value. It just moves against the obstacle until the episode ends and never reaches the goal. The described behaviour would only not occur, if the agent has never experienced the states above him while training, which is very unlikely in our scenario.

Question 3 (i) By introducing the bellmann equation, the agent becomes far-sighted. It now estimates the Q-values by bootstrapping from previous experiences and updates the weights of the network accordingly. It is now able to learn not only from the immediate reward, but also the maximum Q-value of the next state (future rewards). This leads to e.g. the ability to learn that values on the right side of the obstacle have high Q-values even though the distance to goal might be larger, because potential subsequent states from there have very high values. However, we are using a random policy, so it will still be hard for the agent to learn a lot about the states above the obstacle and, therefore, reach the goal.

Question 3 (ii) The loss curve of DQN with target network decreases in the beginning faster and exhibits very characteristic spikes. It decreases faster, because it can easier estimate our target $R + \gamma \max_a \hat{Q}_{\hat{\theta}} (S', a)$ since $\hat{Q}_{\hat{\theta}}$ is not updated after every iteration. With a target network, the network's Q values do no longer continuously rise because the neural network increases Q-values of adjacent states. The spikes in the figure clearly show that $\hat{Q}_{\hat{\theta}}$ is synchronized with the actual Q network every 10 episodes. After each update of $\hat{Q}_{\hat{\theta}}$ we train the Q values with an updated target network that might be very different than before, which increases the loss in the beginning (spike) of each target network update.

Question 4 (i) If ϵ was always 0 (from the beginning), our policy would be deterministic and greedy. Therefore, our agent would always exploit and choose the action with the highest Q value from the beginning. It will follow this policy until it is stuck in a local minimum. Consequently, all following exploration would be unintentional due to updating other states' Q-values through generalization. Through this generalization the agent might follow a new sequence until stuck at a local minimum again. This continues and it is very unlikely that the agent would ever reach the goal state. It might reach the goal state though, if the Q-network's weights were miraculously initialized to lead our agent to the goal in the very beginning.

Question 4 (ii) This might be because the agent has not yet sufficiently explored the states on the left side of the goal. But what it did realize is that on the right side of the goal, the best action is to move left. Now, due to generalization, it might be that the agent generalizes for the states left of the goal, that the highest Q-values are still obtained when moving to the left. Further, moving right, when passing the goal state, is especially difficult to predict due to the discontinuity of suddenly switching directions. A too small network, might not be able to approximate this complex function and, therefore, keep on moving left towards the wall.

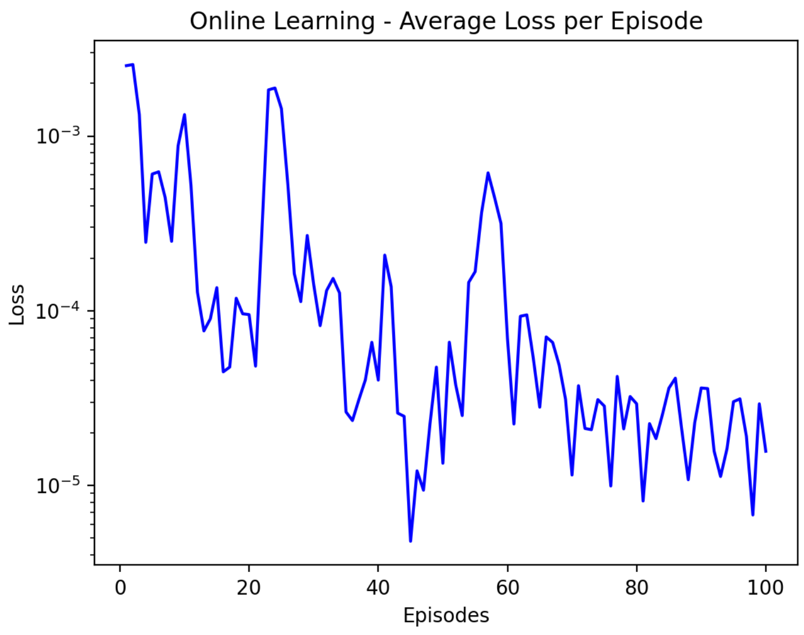

Figure 1 (a)

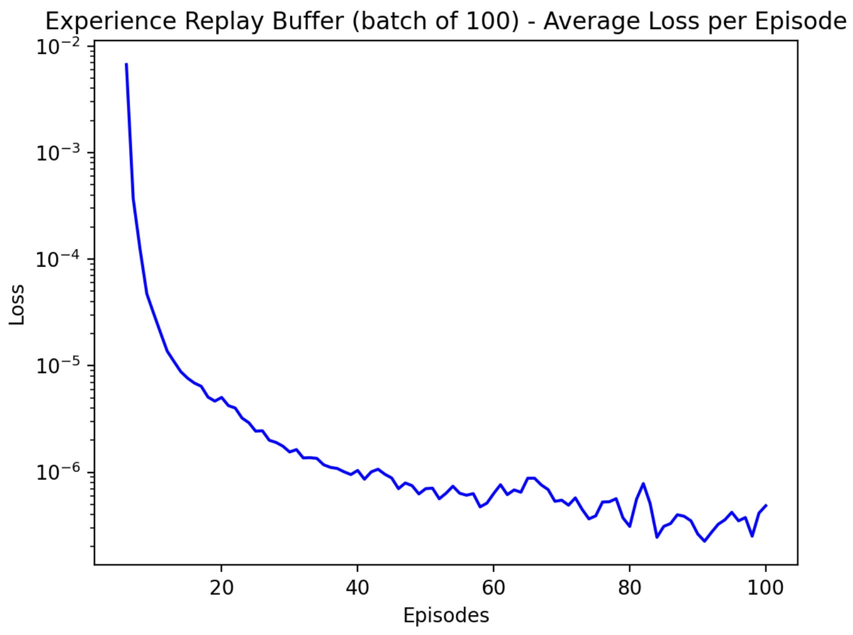

Figure 1 (b)

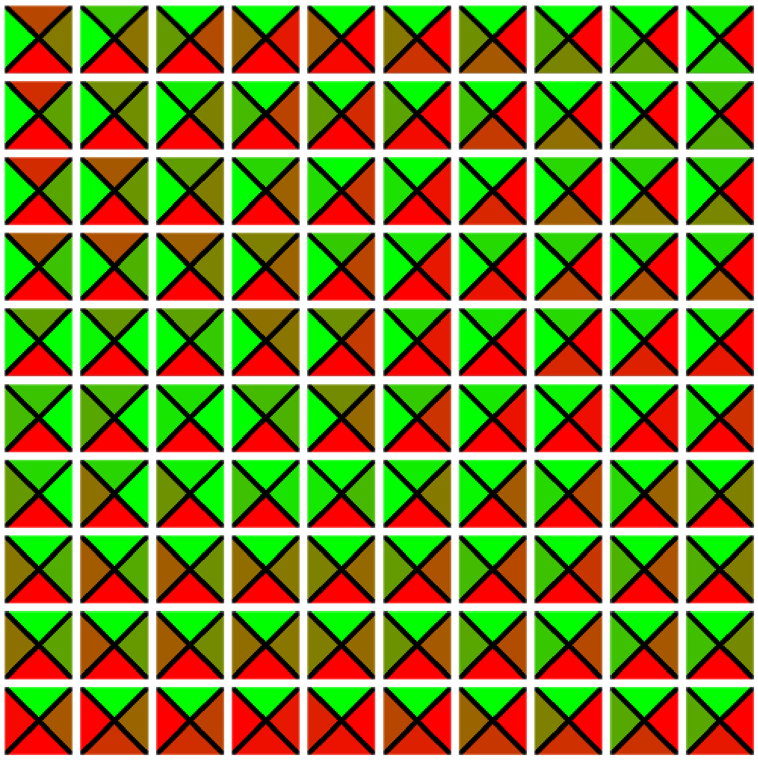

Figure 2 (a)

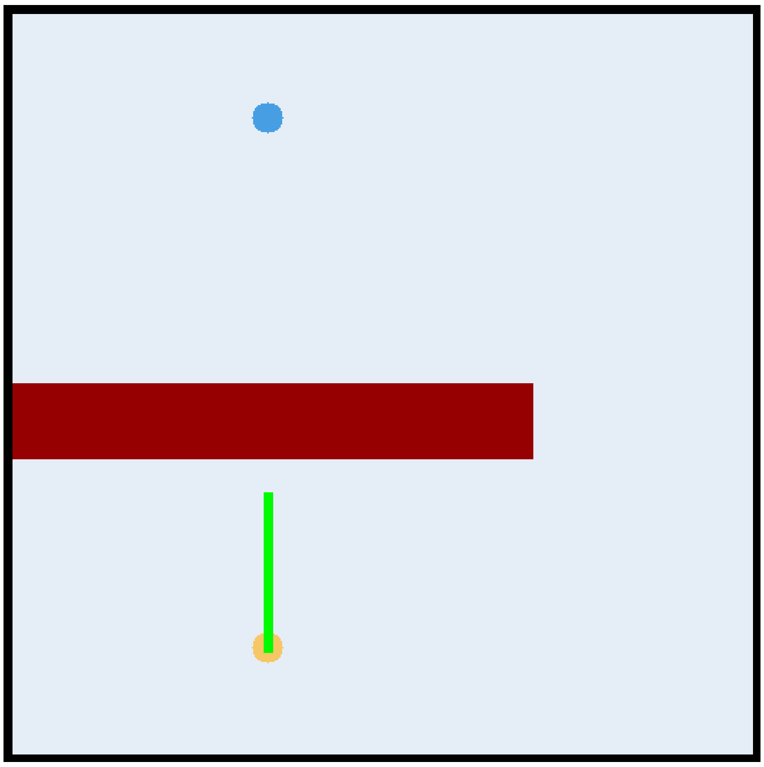

Figure 2 (b)

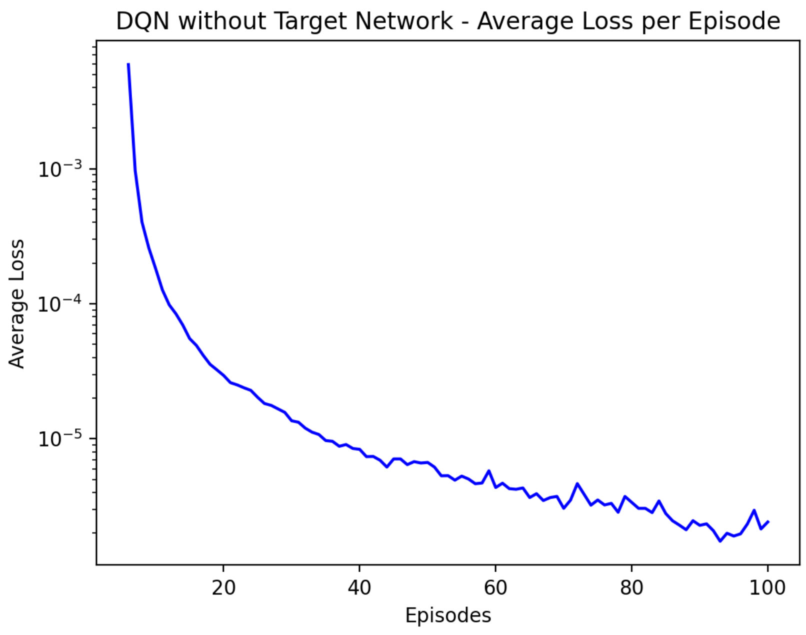

Figure 3 (a)

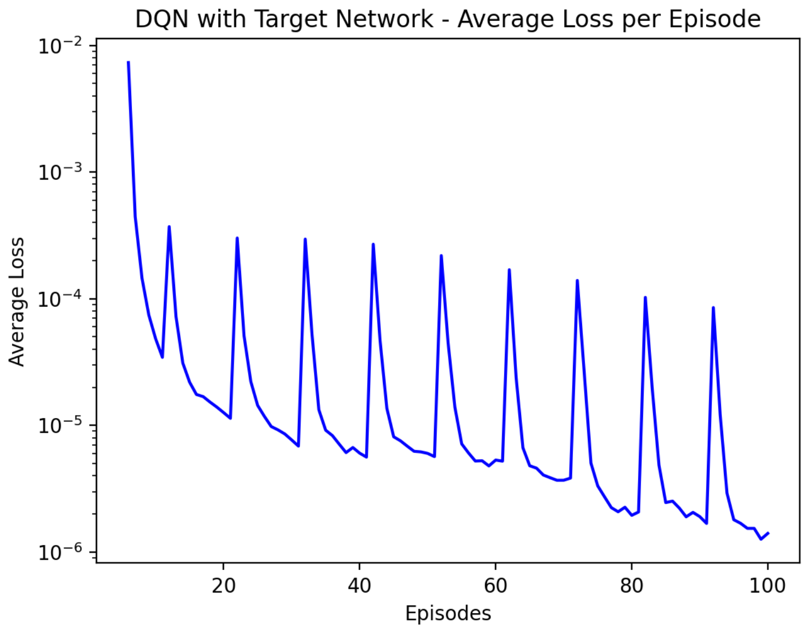

Figure 3 (b)

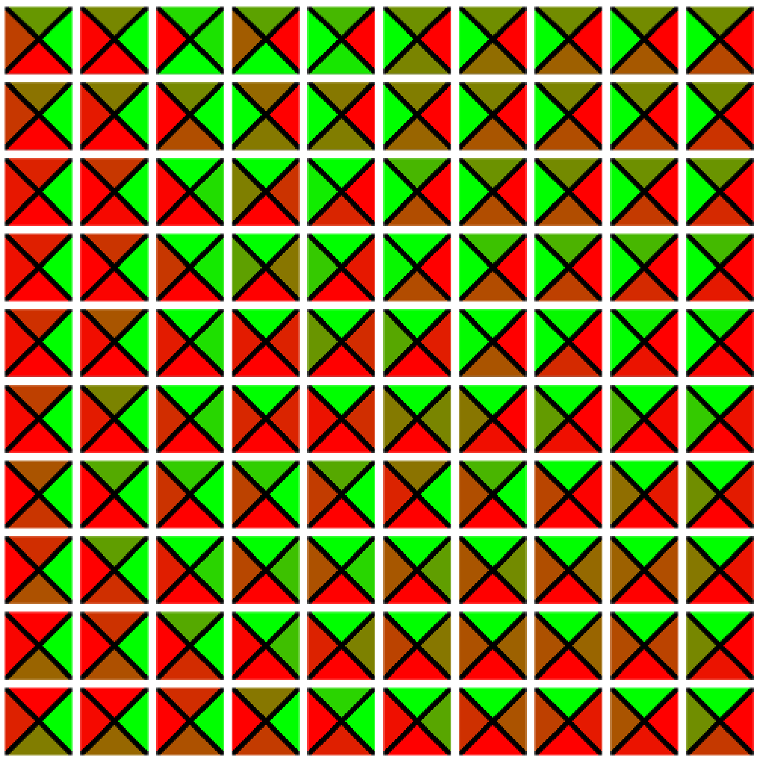

Figure 4 (a)

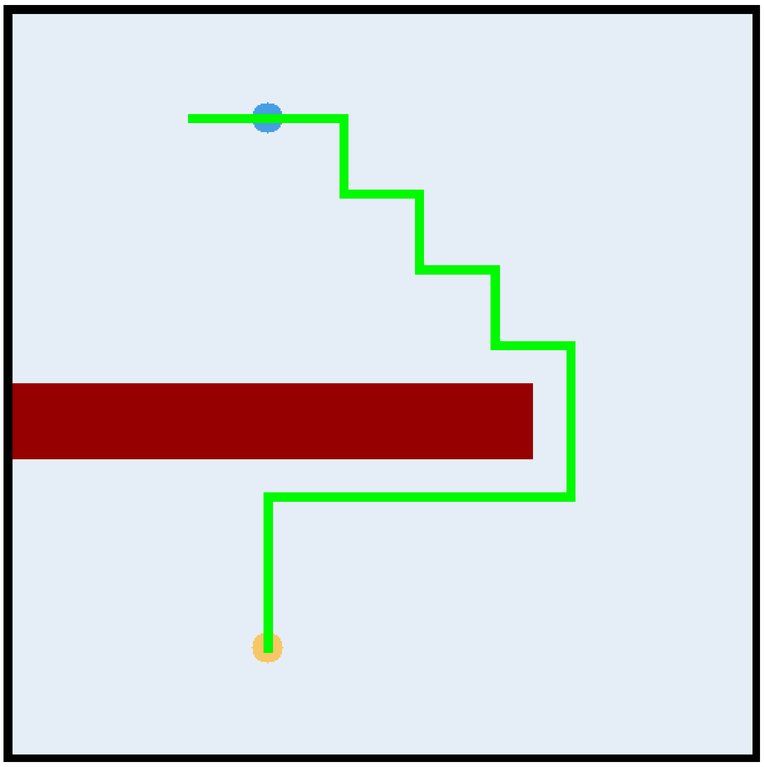

Figure 4 (b)

Description of Implementation for Part 2

My code consists of the following classes: Agent (lines 7-180) includes the implementation of what is called in train_and_test.py; Network (186-212) determines the structure of the neural network; DQN (219-293) is responsible for training the Deep Q Network; and the class ReplayBuffer (299-350) stores and updates the transitions and weights in the buffer. I implemented a Double Deep Q Network with prioritized experience replay buffer and an action space that allows for actions in 6 directions (N,E,S,W and diagonals in forward direction, so NE and SE) (20, 140). The network has 7 layers (and 1 output layer), each with a relu activation function and 128 neurons, except for the first one, which has 256 neurons (191-212). Further the network's learning rate is 0.001, its discount factor is 0.95 and I am using an Adam optimizer (230-232). I made my epsilon and my episode length dependent on the time passed (39,40). The initial episode length is set to 1000 (12) and starts decreasing after 4 min by 50 steps every 14 seconds (112-114). After roughly 8 min the length will stay at 100 (167,168). For the epsilon greedy policy, I am using a decaying epsilon $\epsilon = 1 - 0.9 * (\textit{time elapsed}/480\ \textit{seconds})$ (158). After 8 mins ϵ stays at 0.1. Further, the last 10% of each episode (except episode length is 100), are always focused on exploring, because I set epsilon to 0.9 and the flag exploring to True (119-121). I reset it to the decaying epsilon in the beginning of the next episode if 5 min passed (50,51), otherwise it is reset after 15% of the next episode passed (116,117). The reward function I used is $(1 - distance)^3$ (172-180). Besides incentivizing very short distances, the agent is penalized (reward = 0) for moving against the wall. For training the network, I use a batch size of 256 (304, getting batch through ReplayBuffer method get_batch, 326). Further, I am using a target network, which is synchronized with the Q network every 50 steps (105-108).

In get_next_action() of class Agent (68), a random action is chosen for the first 1000 steps (self.exploration_steps, line 24). After 1000 steps, the action with the highest Q-value is determined (75) by the instance method get_highest_q_value_and_action() (146-151) and, based on this action, an epsilon-greedy-policy chooses the next action through get_epsilon_greedy_action() (155-163). Once the method set_next_state_and-_distance() is called, the current reward is calculated (91, 172-180). With each step taken, a new transition is added to the buffer and its weight is initialized to the current maximum weight (93-98). When training the network (103, 240-252) with a sample batch of size 256, which is drawn according to the priorities of the replay buffer, the loss is calculated (256-288) according to the double Q learning algorithm (269-273). Also I update the weights for the prioritized experience replay buffer according to the difference between the target tensor and Q(s,a) (282-285). When updating the weights (346-350), I add an ϵ of $0.2 * max(weights)$ to ensure that there is a minimum probability of training this Q-value.

I also implemented early stopping: If the episode length is 100, the agent will try out a greedy policy every three steps and in the last minute even every other step (123-126). If the agent reaches the goal under the greedy policy, self.stop becomes True (128,129) and the network is no further trained (102, self.new_greedy_round stays True). In case it does not reach the goal, self.stop stays False and training continuous with epsilon of 0.1 (53-55).