Deep RL ex.1

Sagiv Yaari [300553500]
Hadar Sharvit [208287599]

November 30, 2022

1 Tabular Q-Learning

1. In Value-iteration, for example, we have seen that the update rule is of the form

$$V(s) \leftarrow \max_{a \in A} \sum_{s',r} Pr[s', r|s, a]\big[r + \gamma V(s')\big] \tag{1.1}$$

   Without a model of our environment, we do not have access to the probability space $Pr[s', r|s, a]$, hence the update rule cannot be calculated.

2. In model-free methods we usually only sample the environment, without actually using the induced probabilities. This means that instead of using an update rule that explicitly mentions the given probabilities, we generate a trajectory and use that trajectory to infer the reward / update rule / etc...

3. While SARSA is an On-Policy TD control algorithm, Q-Learning is an Off-Policy one. In other words, Q-Learning is based on a greedy approximation of the optimal policy (which is the behaviour policy), and SARSA uses only the current policy.

4. When using an ε-greedy approach, we enforce our agent to explore the environment, allowing it to consider actions that are sub-optimal but have the potential of resulting in a larger return over the long run. Rephrasing the above - ε-greedy approach balances both exploration and exploitation, which is preferred over exploitation alone.

5. Python script:
   we used the following set of hyper-parameters:

$$\alpha = 0.25,\ \gamma = 0.95 \ and\ \varepsilon(t) = \begin{cases} 1.0, & \text{if } t = 0 \\ \varepsilon(t-1) - \frac{1}{N}, & \text{otherwise} \end{cases} \tag{1.2}$$

   where N is the number of episodes (5000) and ε is defined using linear decay.
   Below are the required plots

2 Deep Q-Learning (DQN)

1. As every state s_{t+1} is highly correlated to the previous state s_t, learning from consecutive samples means that the training process would have high variance. By randomly

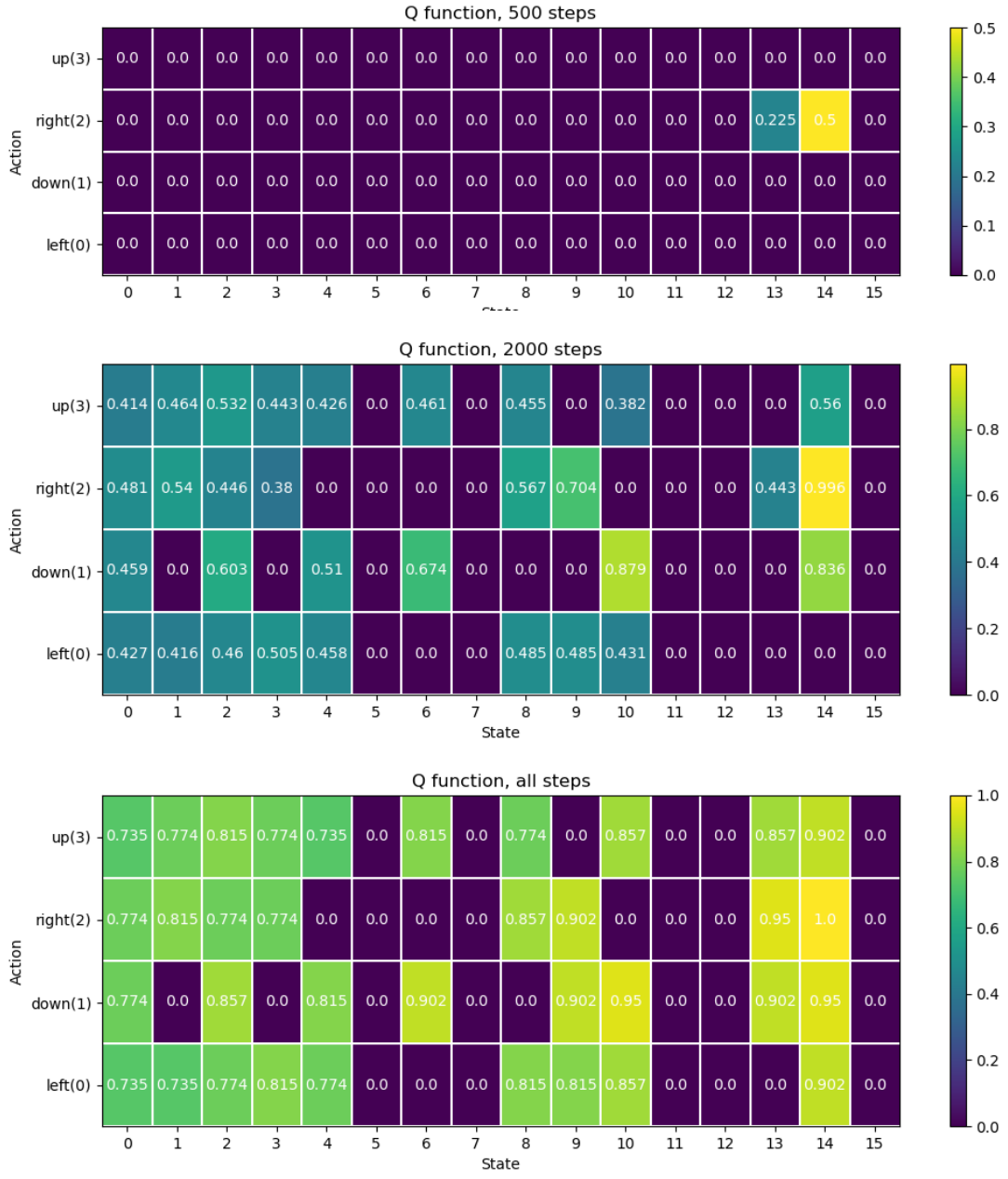
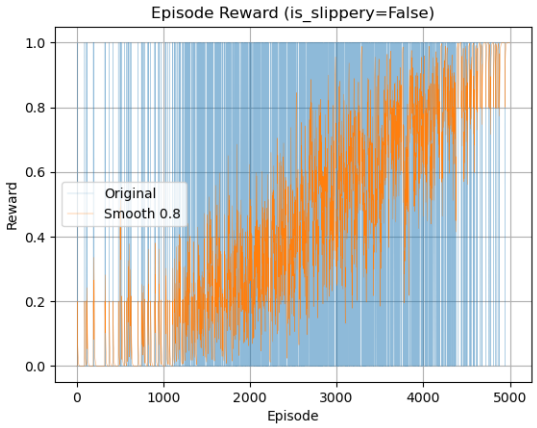
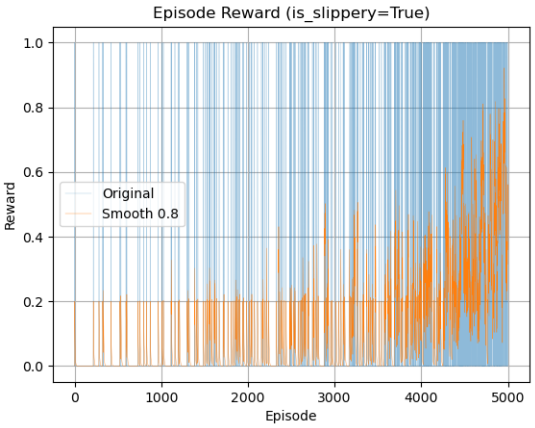


Figure 1: A color map describing the Q-function over time. We can see the gradual increase in value for important state-action pairs. For example - moving right (2) from 14 is a guaranteed win, hence $Q(a = 2, a = 14) = 1$. Note that all columns with zero values represents Hole states (H)



(a) Deterministic environment



(b) Random environment

Figure 2: Episode reward for FrozenLake-v1. As the reward is binary, a smoothed reward is more informative - we notice the upward trend over time, reaching values closer to 1 in (*a*), and to around 0.8 in (*b*), which is a more difficult environment

(a) Deterministic environment

(b) Random environment

Figure 3: The average number of steps needed to reach the goal over 100 consecutive episodes. In cases where the goal was not reached withing $n\_steps = 100$ (either by falling into $s = Hole$ or simply by wandering about the environment without reaching $s = Goal$), the value was set to 100

sampling the experience replay we decorrelate the samples hence gaining a much more stable training process.

2. By fixing the weights of our target network for a period of time (C), we limit the "damage" that random over estimations may have on the learning process. Keeping the network constant like so will make sure that outliers state-action pairs would not affect the training process as drastically, hence allowing for a more stable process.

3. Python script:
   We used the following set of hyper-parameters:

```
{"hidden_dims": {"shallow": [32,32,32],
                 "deep": [16,32,32,16,16]},
"lr": 0.01,
"min_lr": 0.001,
"lr_decay": 0.999,
"epsilon_bounds": [1.0,0.2],
"eps_decay_fraction": 0.25,
"gamma": 0.9999,
"learning_epochs": 16,
"batch_size": 128,
"target_update_interval": 16,
"steps_per_epoch": 128,
"buffer_size": 2048,
"min_steps_learn": 2048,
"inner_activation": "relu",
"final_activation": "relu",
"optimizer_name": "Adam",
"loss_fn_name": "mse",
"dropout": 0.1}
```

   More specifically, we saw that the parameters that had the largest impact on the training process were:

   - The hidden layers: as our network increased in size, it was able to better understand the optimal state-action value, hence the reward performance was better than a shallower layer.
   - epsilon: It was prominent that higher ε values at the beginning and smaller values later on in the training process were very important. Initially, our agent was able to explore the environment further when it needed to, and later in the training process, was also to able exploit the environment when a proper Q-function estimator was formed.
   - Experience replay buffer size: we understood that choosing too large of a buffer was not optimal, as sampling very "old" experiences meant training our agent on a highly non optimized targets.
   - target update interval: a trade-off between stable targets and optimal targets had to be made, as on the one hand, updating our target network frequently means unstable training process, yet on the other hand, low frequency update means that

our targets are far from optimal, as they are based on small number of experiences updates.

Using a shallow network, it took about 7000 episodes to obtain an average reward of at least 475 over 100 consecutive episodes. For the deep neural network, it only took about 4300. In the following figure we have added the mean reward and loss values for three networks - a shallow DQN, a deeper DQN and a double DQN.

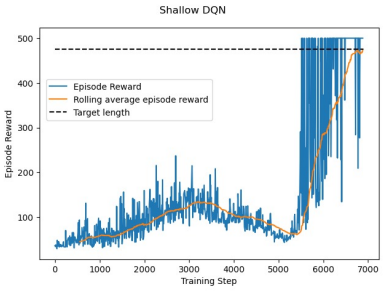


Figure 4: Shallow DQN

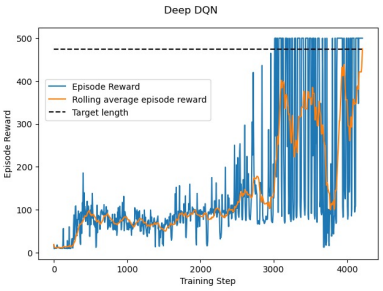


Figure 5: Deeper DQN

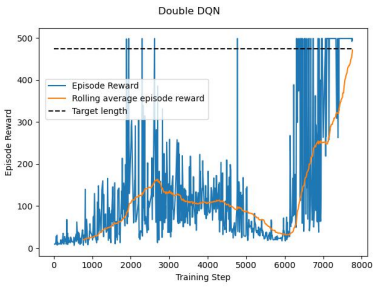


Figure 6: Double DQN

Figure 7: Episode reward for CartPole-v1. We can see How a deep neural network was able to achieve the same results after only 4000 episodes compared to moer than 7000 for a shallow one. Furthermore, a Double DQN approach was only able to reach the needed goal after more than 8000 episodes, which is reasonable as it is not overestimating the target values.

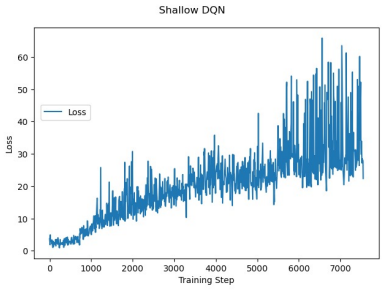


Figure 8: Shallow DQN

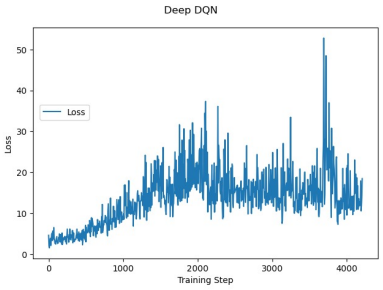


Figure 9: Deeper DQN

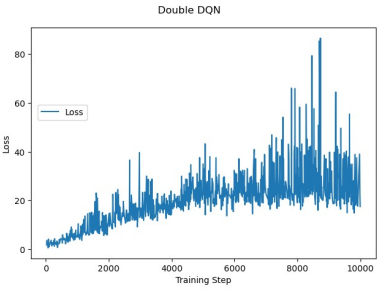


Figure 10: Double DQN

Figure 11: Loss plots for CartPole-v1. We can see that all of our models loss generally increases. As this may not be reasonable in supervised learning, it is not unusual in RL. One reason for that may be that compared to the beginning of the training process, where few state-action pairs had any value at all, in future episodes many such pairs may become valuable. When taking the difference is between the new Q function and a rather old version of it, the difference in values may be large, which contributes to the loss. This also explains some of the variance we see, which is the result of our target being changed over and over again.

3 Improved DQN

We've used a version of the Double-DQN network: notice how in the DQN algorithm we both choose an action a_t using $\hat{Q}(s_t, a, \theta)$ and evaluate it (when calculating y_i), which means that we tend to overestimating the target values. To address this issue we replace the current update of y_i (when not $done_i$) with

$$y_i^{DoubleDQN} = r_i + \gamma \hat{Q}\left(s_{i+1}, argmax_{a' \in A} \hat{Q}(s_{i+1}, a', \theta), \theta^-\right) \tag{3.1}$$

In other words, we choose an action ($argmax$) using a network with parameters θ (that is currently being trained), but evaluate the action using a network with parameters θ^- (that is not being trained).
We've used the same hyper parameters as for the shallow DQN (see previous section), and added the plots in the previous section as well. While our DQN model over estimates the value of the state-action pairs, the DDQN model accurately evaluates those. This means that in the long run we can anticipate the reward to decrease for the regular DQN but remain rather stable for the DDQN.