Deep Reinforcement Learning Exercise 1

Shahaf Yamin 311530943
Ron Sofer 204738637

December 4, 2021

Running Instruction for the script of this assignment.

- Question 1:
  run the following command -

  ```
  python q_learning.py
  ```

- Question 2:
  run the following command -

  ```
  python DQN.py --Q 2 --exp [exp_name]
  ```

  to receive results under exp_name folder in the log-directory.

- Question 3:
  run the following command -

  ```
  python DQN.py --Q 3 --exp [exp_name]
  ```

- Extras (DQN):
  run the following command -

  ```
  python DQN.py --Q 2
  ```

  with the following optional arguments to change configurations.

  - "–model_layers 5" for 5 layers model
  - "–model_type DDQN" to use Double DQN model
  - "–buffer_type priority" to use Prioritized Experience Replay
  - "–exp [exp_name]" to set experiment name
  - "–n_episodes [integer]" to set the number of max episodes
  - "–no_breaks" to force running until max number of episodes
  - "–reward_shaping" without reward-shaping explained in Section 3.1
  - "–reward_shaping 1" to add RS_t^1
  - "–reward_shaping 2" to add RS_t^2
  - "–reward_shaping 1 2" to add RS_t^1 and RS_t^2

1 Tabular Q-learning

1.1 Why methods such as Value-Iteration cannot be implemented in such environments?

Value-Iteration and Policy-Iteration methods assume knowledge of the environment's dynamic. Therefore, in case that we don't know the underline environment dynamics, we can't find the optimal policy or value function by using these methods.

$$V_{k+1}(s) \leftarrow max_{a \in A(s)} \sum_{s' \in S, r \in R} p(s', r|s, a)[r + \gamma V_k(s')] \tag{1}$$

For example, in the above equation, we can see that the iterative update of the Value function in the Value-Iteration algorithm uses the underline dynamics distribution p, which is unknown in this scenario.

1.2 How do model-free methods resolve the problem you wrote in previous question?

Model-free methods uses only their experience with the environment and the feedback returned from such an experience, i.e., they perform an action based on the observed state and they receive an immediate reward and the next-state. Thus, they don't need the knowledge of the environment underline dynamics for their update-rule. For example, in SARSA, the update rule is as follows

$$Q_{k+1}(S, A) \leftarrow Q_k(S, A) + \alpha[R + \gamma Q_k(S', A') - Q_k(S, A)] \tag{2}$$

Notice that there is no use in the dynamics, but only in the immediate reward, the current and next state, the current and next action, and the current approximation of the action-value function $R,\ S,\ S',\ A,\ A'$, and $Q_k(\cdot, \cdot)$ respectively.

1.3 What is the main difference between SARSA and Q-learning algorithms?

The main difference between SARSA and Q-learning is that SARSA is an on-policy, while Q-learning is an off-policy method. In other words, SARSA uses the behavioral policy π for choosing the next action A', and also for updating the action-value function Q. Whereas in the Q-learning algorithm we use the behavioral policy π for choosing actions, but for updating Q we use a different policy μ (the greedy policy), which can be seen in the update equation as $max_a Q(S', a)$.

1.4 Why is it better than acting greedily (choosing an action with $argmax_a Q(S', a)$)?

Since the action-value function is only an estimation of the true return and not the actual return.

$$\hat{Q}_\pi[s, a] \approx \mathbb{E}^\pi[G_t | S_t = s, A_t = a] \tag{3}$$

Acting greedily may result in a wrong solution and prevent convergence to the real action-value function. Using sampling methods such as $\epsilon - greedy$ may ensure enough exploration of the state action pairs and therefore in many cases ensures convergence. Usually we will use GLIE methods (Greedy in the Limit with Infinite Exploration), which ensures exploration, but in the limit they achieve a greedy policy.

1.5 Python Script

In this section we solved FrozenLake-v1 environment using Q-Learning algorithm.

1.5.1 Problem Definition

Given a penguin on a frozen lake, which is described by a 4x4 grid with holes and a goal state (fish), both defining terminal states. For every transition the penguin gets a reward of 0, unless the penguin reaches the goal states and than he receives a reward of 1. Mathematically speaking, we can define the reward function as followed:

$$R_{t+1} = \begin{cases} 0\,, S_{t+1} \neq 15, \\ 1\,, S_{t+1} = 15, \end{cases} \tag{4}$$

The penguin can take four possible actions, $\mathcal{A} = \{Left, Down, Right, Up\}$, but the surface is slippery and only with probability $\frac{1}{3}$ the penguin will go into the intended direction and with probabilities $\frac{1}{3}$ to the left and $\frac{1}{3}$ to the right of it. It is assumed that the boundaries are reflective, i.e., if the penguin hits the edge of the lake it remains in the same grid location. Denote by $\mathcal{S}$ the state space and let $\mathcal{R}$ be the set of all valid rewards {1, 0}.

1.5.2 Hyper-Parameters Tuning

As part of this section, we will describe our process of finding a good combination hyper-parameters for the Q-learning algorithm. To evaluate performance of this algorithm for different values of hyper-parameters we have measured the algorithm performance in terms of the average duration of the episode (T) and the return (G_t). We tuned three hyper-parameters, the discount-factor γ, the learning rate α, and the exploration decay rate ϵ_D which is shown in Eq.5. In order to calculate the estimated average we have trained the agent 10 different times (Monte-Carlo technique), and averaged the results of the different experiments. We found our combination of hyper parameters in an iterative process,

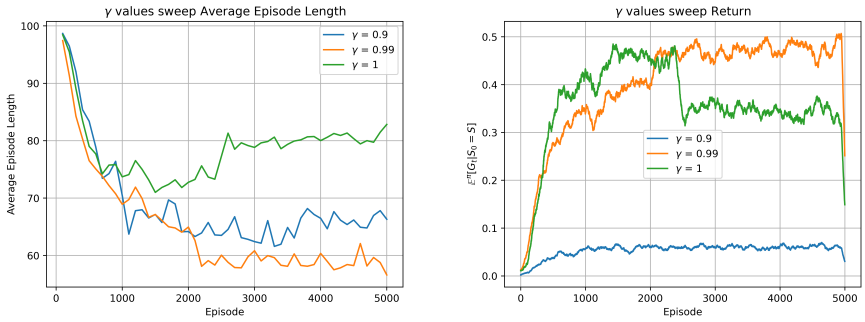
first we've fixed both α and ϵ_D parameters, then trained the agent over different values of γ. The graphs of T and G_t as function of the episode number over different discount-factors can be seen in Fig.1. Second, we found the optimal learning rate α for the fixed best γ found earlier, and the same ϵ_D. The graphs of T and G_t as function of the episode number over different learning-rates can be seen in Fig.2. Finally, we used different exploration decay rates to find the optimal ϵ_D with the best learning-rate and discount-factor. The influence of the parameter ϵ_D can be seen in the equation

$$\epsilon(s) = \frac{1}{N(s) * \epsilon_D + 1} \tag{5}$$

When $N(s)$ equals to the number of visits in state s. ϵ defines the probability in which we choose a random action due to the epsilon-greedy policy, the use of $N(s)$ ensures that when reaching an under-visited state we will tend to explore. The graphs of T and G_t as function of the episode number can be seen in Fig.3. From the graphs we can suggest that the best hyper-parameters for this task are $\gamma = 0.99,\ \alpha = 0.5,\ \epsilon_D = 0.05$.

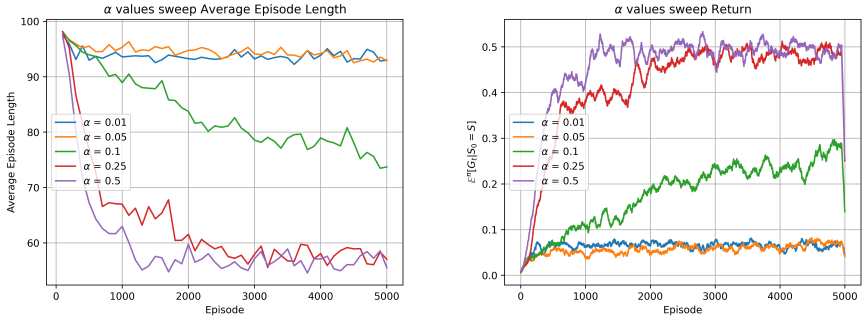
1.5.3 Results

In this section we will present our algorithm final results under this task setting with the aforementioned hyper-parameters that we've found in the previous section. In Fig.4 we present the action-value function $Q(s, a)$ after 500 episodes, 2000 episodes, 5000 episodes in a normal training scheme. We also show the "Optimal $Q^*(s, a)$" which is achieved by training for 100,000 episodes with appropriate hyper-parameters and exploring starts, i.e., in every episode we start in a random non-terminal state. We suggest this "Optimal $Q^*(s, a)$" is a good-enough approximation of the optimal action-value function $Q^*(s, a)$. We justify this suggestion by using the fact that the convergence of the Q-learning algorithm is guaranteed under this scenario.
We also present both greedy policies induced by the approximated action-value functions after 5000 episodes and after 100,000 episodes in Fig.5. One can suggest that the policies are not the same, since in state 6 we choose action 'Left', while the "optimal" policy chooses 'Right', but by understanding the problem we can confidently say these two actions have the same value from state 6.
In Fig.6 we show the number of steps which we visited in each state, we note that under this scenario the 'brightest' path is also the optimal path, since in each state the agent can perform an action without any chance of falling into a hole.



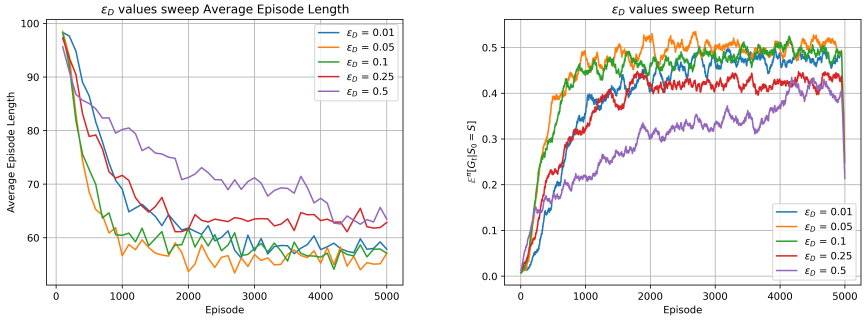
Average Episode Length Average Return

Figure 1: γ Sweep



Average Episode Length Average Return

Figure 2: α Sweep



Average Episode Length Average Return

Figure 3: ϵ_D Sweep

Figure 4: $Q(s, a)$ as heat maps

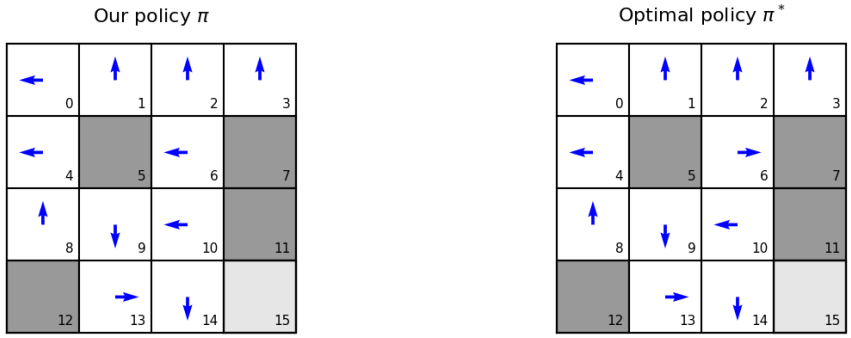


Figure 5: Policies π and π^*

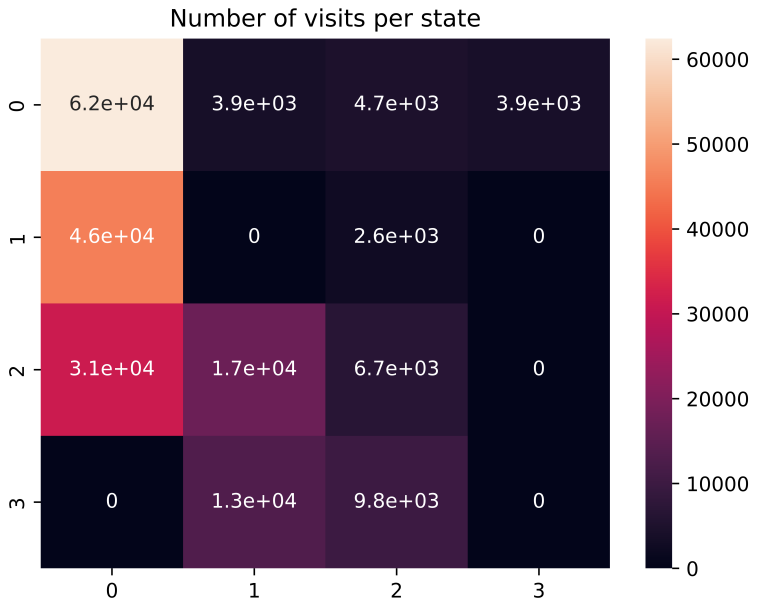


Figure 6: Number of visits for each state

2 Deep Q-learning

2.1 Experience Replay - Why do we sample in random order?

When training a neural network, we use Stochastic Gradient Descent method in order to minimize our loss function with a noisy gradient estimation. This estimation tries to avoid convergence to local optimums regimes in our parameter space while preserving the true gradient direction on average. When estimating our gradient, we assume that our training samples are drawn from an i.i.d distribution. This i.i.d assumption (that is crucial for Neural Network training) does not hold because there is a high correlation between consecutive samples in a given trajectory. The solution for this problem is to use a Replay Buffer. The replay buffer with a capacity N, stores transitions $(S_t, A_t, R_{t+1}, S_{t+1})$ and in each training session, a batch of size $|B|$ is uniformly sampled from the buffer. In contrast to consuming samples online and discarding them after that, uniform sampling from the stored experiences means they are less heavily correlated and can be re-used for learning. Usually, the replay buffer has a fixed capacity of size N, and it's stores transitions using First-In-First-Out(FIFO) queue management policy.

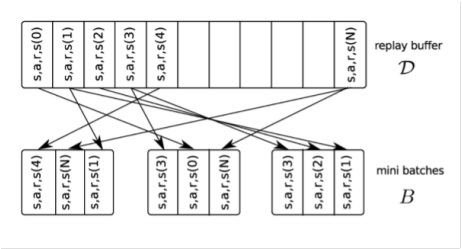


Figure 7: Replay Buffer with capacity N, and mini batches the sampled from it.

2.2 Target Network - How does this improve the model?

In the naive DQN implementation, we use a single network for both target and policy networks, i.e. $\theta' = \theta$, which may cause instability issues. when using a single network, we get an estimation of $q_\pi(S_t, A_t)$ which depends on the same parameters θ as the parameters of $\hat{q}(S_t, A_t; \theta)$. On the contrary, in supervised learning, the target should be independent of the network parameters, and in our case, the target (the estimation of $q_\pi(S_t, A_t)$) depends on the parameters θ. The problem behind the naive implementation, is that we adjust the parameters in order to push $\hat{q}(s, a; \theta)$ towards the target $\hat{y}$. This procedure likely to cause changes in the target $\hat{y}$ as well because it also depends on same parameters θ. This case usually causes instability because it is like "chasing our own tail."

The solution is to define two networks; the policy network $\hat{q}(s, a; \theta)$, and the target network - $\hat{q}(s, a; \theta')$. The key idea is to create a "periods of supervised learning" by using the target network $\hat{q}(s, a; \theta')$ for the target calculation. The target network is a copy of the policy network, and it is updated periodically, i.e., every $C \in \mathbb{N}$ iterations we update the target network parameters with the policy network parameters $\theta' \leftarrow \theta$. Following this method, the target values are fixed for some period C, and thus during these periods, the policy network training procedure acts like supervised learning.

2.3 Python Script

We were asked to solve a control problem with continuous state and discrete action spaces using Deep Q-Network(DQN) in this question.

2.3.1 Problem Definition

The environment is the CartPole environment. The goal of CartPole is to balance a pole connected with one joint on top of a moving cart. We define the state space $\mathcal{S} \subseteq \mathbb{R}^4$ where each state $s \in \mathcal{S}$ contains the following information $s =$ [cart position, cart velocity, pole angle, pole angular velocity]. where each dimension belongs to the following sets:

- *cart position* $\in [-4.8, 4.8]$
- *cart velocity* $\in [-\infty, \infty]$
- *pole angle* $\in [-0.418, 0.418]_{radians}$
- *pole angular velocity* $\in [-\infty, \infty]$

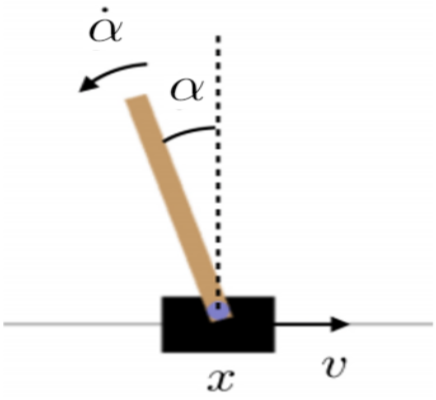


Figure 8: CartPole Illustration.

The action space $\mathcal{A}$ is discrete $\mathcal{A} = \{push\ cart\ to\ the\ left, push\ cart\ to\ the\ right\} = \{0, 1\}$.
The reward is +1 for every step taken. An episode is terminated if:

- The pole angle goes beyond ±12 degrees or ±0.2094 rad.
- The cart position is larger than 2.4.

We define the score as the total reward (= total number of steps made) received in one episode. We evaluate our agent overall performance by using a metric which we term as "avg score", where we define "avg score" as the average score over 100 consecutive episodes. Our goal is to maximize the "avg score" value. Throughout the rest of this work, we have chosen to use RMSProp optimizer. As our exploration policy we use $\epsilon - greedy$ with the following update rule:

$$\epsilon \leftarrow max\Big(e^{-(\frac{N_E}{\sigma})^2}, 0.01\Big) \tag{6}$$

Where N_E is the episode number and σ is a hyperparameter that determines the smoothness of exploration decay. We set $\sigma = 650$ which means that at episode 1300 we receive $\epsilon \approx 2\%$ to perform a random action. The graph of ϵ vs N_E is shown in Fig.9.

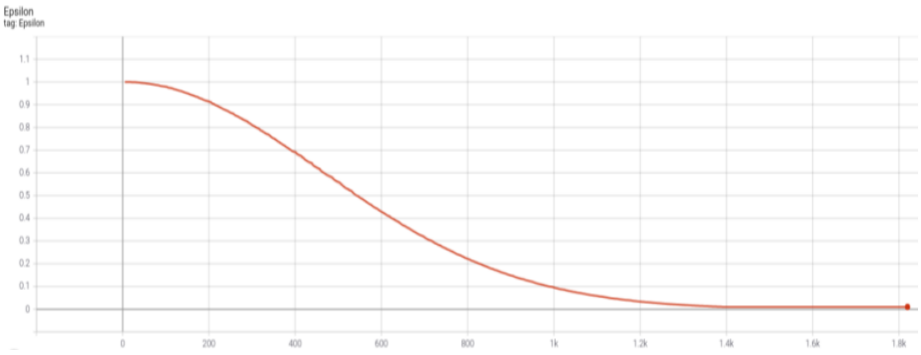


Figure 9: ϵ as function of the episode number

2.3.2 Hyper-Parameters Tuning

As part of this section, we will describe our process of finding a "good enough" combination of hyper-parameters for the DQN networks. To evaluate performance of this algorithm for different hyper-parameters values, we have measured the algorithm performance in terms of "avg score". We have tuned seven hyper-parameters, the discount-factor γ, the learning rate μ, the smoothness of exploration decay σ which is shown in Eq.6, the batch size $\mathcal{B}$, number of hidden layers M (and the layers' sizes), update frequency of the target network K and the replay buffer capacity C (We use First-In-First-Out(FIFO) technique after the replay buffer is full). Due to the high dimension of the

hyperparameter space, it is not feasible to scan all the available combinations. Furthermore, training a model until convergence is time-consuming, and computational expensive, therefore we selected the most interesting experiments to compare between themselves. We have visualized the loss at each time-step and the "avg score" at each episode using TensorBoard. In our search for a good combinations of hyperparameters, we have encountered with some unstable results, and our main conclusion is that stability has the most important role in training such agent. Fig.10 shows two pairs of identical training paradigms when one pair with 3-layers network and the other with 5-layers, if we look at the 5-layers networks we can see that one network achieved the goal, while the other didn't. These results are what we call unstable training paradigm. Our best combination of hyperparameters are shown in Table 1.

Table 1: Hyperparameters values

γ	μ	σ	$\mathcal{B}$	M	hidden-layers sizes	K (episodes)	C
0.95	0.00025	650	256	3	(8,8,4)	50	100,000
				5	(16,16,8,8,4)		

2.3.2.1 Experiments

Since stability has the most important role in our opinion, we sort the hyperparameters' importance by their influence on the training stability. The most important hyperparameter for this environment, is the network size, i.e., the depth and width of network, we saw that a combination of shallow and narrow network produced more stable training procedures. Therefore, we decided that using a 3-layer model with narrow layers is preferred, as shown in Fig.10. The second important hyperparameter, is the batch-size combined with buffer-capacity, we concluded that using a larger batch-size, achieves a higher quality of gradient estimation, which in turn improves network stability and allow us to use a bigger step size. A large buffer-capacity reduces correlation inside the batch, getting us closer to the i.i.d assumption, thus the experiences in the batch are more suited for this type of training. The influence of the pair $(\mathcal{B}, C)$ is shown in Fig.11. Third, the target-network update frequency (K) plays an important role. Updating too often may resolve in 'chasing our own tail' and divergence, while seldom updating will probably lead to poor performance. In Fig.12 we present the importance of K with three similar experiments with the only difference of the value of K. We compare the avg score and loss to present the divergence of the unstable updates, with $K = 1$ we can see a quick divergence, while with $K = 500$ the loss increases but slower, whereas with a balanced K, $K = 50$ we get a convergence.

2.3.2.2 Visual Comparison

Throughout all of the experiments shown in this section, we used the Huber loss since it serves our goal of combating with the instability issue. We have also

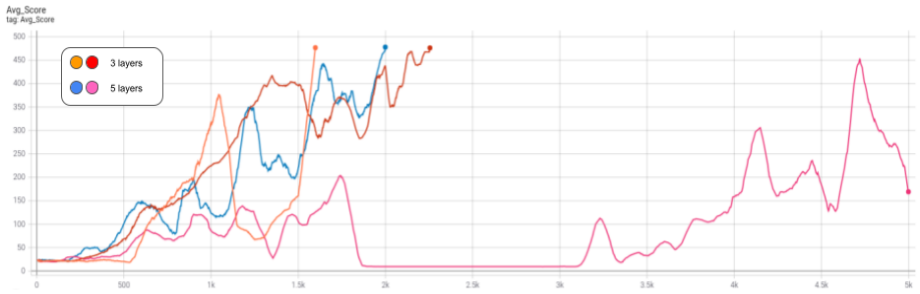


Figure 10: Avg-Score of 3-layer and 5-layer networks during training

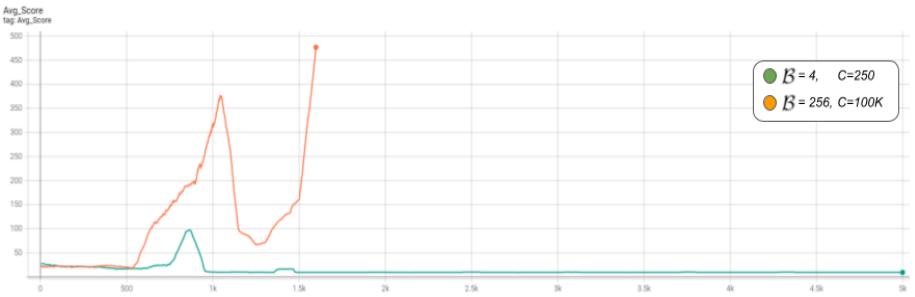


Figure 11: Avg-Score with different $(\mathcal{B}, C)$ pair

used the same σ, γ, μ in order to evaluate this phenomenons without any biases.

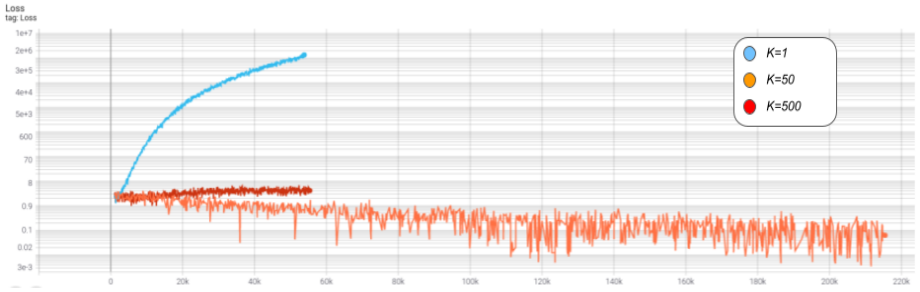
2.4 DQN Results

In Fig.13 we show the score over episodes, and the loss over the time steps of the DQN model during training. We can observe the learning curve of the agent in both loss and score graphs. The number of episodes until convergence of best the 3-layers network is 1,600.

3 Improved DQN

As part of this section we will evaluate the importance of three different methods which aim to improve the performance of the vanilla Deep Q-network. We have chosen to evaluate performance of "Reward-Shaping", "Double DQN", and "Prioritized Experience Replay" methods. First, we will begin by describing each method purpose and afterwards, we will proceed for examination of each method's impact over the agent's performance. The hyperparameters used for the experiments in this section are those presented in Table 1.

Avg-Score



Loss

Figure 12: Avg-Score and Loss for different K values, y-axes are in logarithmic scale

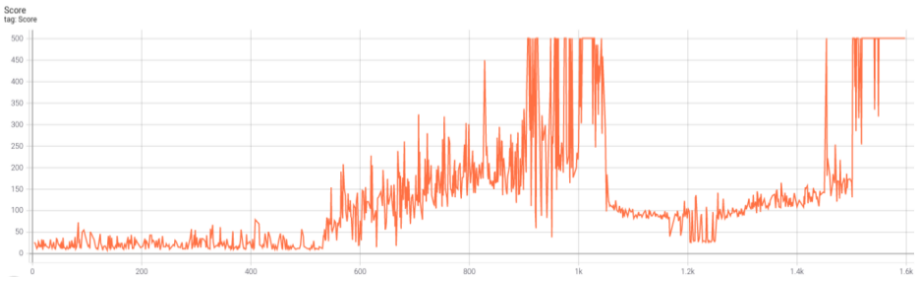
3.1 Reward Shaping

The reward signal is crucial in the training progress, since it is the only way to estimate weather we did good or bad. Receiving a reward of 1 for every step in which the agent didn't lose is considered not very informative, therefore we suggested to shape the reward signal in a way that may lead our agent to a better understanding of what is considered good, and bad. We formulate two types of reward shaping RS^1, RS^2 which is formulated in equations [7,8] and we compare the performances of using the give reward $\tilde{R}_t = R_t$, using only RS_t^1 , i.e., $\tilde{R}_t = R_t + RS_t^1$, and both RS_t^1, RS_t^2, i.e., $\tilde{R}_t = R_t + RS_t^1 + RS_t^2$. These comparisons are shown in Fig.14

$$RS_t^1 = -0.5 \cdot \left(\frac{|\alpha|}{\alpha_{max}} + \frac{|x|}{x_{max}} \right) \tag{7}$$

$$RS_t^2 = -100 \cdot \mathbb{1}_{(S_{t+1} \ is \ terminal) \ \& \ t<499} \tag{8}$$

Where we denote $\mathbb{1}_P$ as the indicator function of P, α as the pole angle, and x as the cart location, both are normalized with their respective maximum value. The logical reason is that we aim to find a policy that stabilises the poll angle while preserving the cart in the center of the x axis ($x = 0$). We consider $(\alpha, x) = (0, 0)$ a safer point comparing to other values for this pair. Thus, we



Score

Loss

Figure 13: Score and Loss of the chosen hyperparameters

used RS_t^1 to point our agent towards the this behaviour by giving it bigger reward. In addition, RS_t^2 goal is to punish the agent upon losing the game (winning is considered only when achieving highest possible score of 500).

3.2 Double DQN

The popular Q-learning algorithm is known to overestimate action values under certain conditions. The reason for this overestimation is the max operator in standard Q-learning and DQN, which uses the same values both to select and to evaluate an action. This makes it more likely to select overestimated values, resulting in overoptimistic value estimates. To prevent this, we can decouple the selection from the evaluation. This is the idea behind Double Q-learning. We can decouple the selection by using two action-value functions, such that there are two sets of weights, θ and θ'. For each update, one set of weights is used to determine the greedy policy and the other to determine its value. For a clear comparison with the vanilla DQN we will show the estimated value in both cases,

$$Y_t^{DQN} = R_{t+1} + \gamma \cdot \max_{a \in \mathcal{A}} Q(S_{t+1}, a; \theta_t') \tag{9}$$

$$Y_t^{DoubleDQN} = R_{t+1} + \gamma \cdot Q(S_{t+1}, \arg\max_{a \in \mathcal{A}} Q(S_{t+1}, a; \theta_t); \theta_t') \tag{10}$$

Notice that the selection of the action, in the argmax, is still due to the online weights θ_t. This means that, we are still estimating the value of the greedy policy

according to the current estimation of the action values. However, we use the second set of weights θ'_t to fairly evaluate the value of this policy.

3.3 Prioritized Experience Replay

Prioritized Experience Replay (PER) is a conceptually straightforward improvement for the vanilla Deep Q-Network (DQN) algorithm. Prioritized Experience Replay, as the name implies, will weigh the samples so that "important" ones are drawn more frequently for training. Our algorithm goal is to find a set of parameters $\theta^\star$ such that,

$$\theta^\star = \arg\min_\theta \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim\mathcal{B}}[(r_{t+1}+\gamma\cdot\max_{a\in\mathcal{A}} Q(s_{t+1},a;\theta') - Q(s_t,a_t;\theta))^2] \quad (11)$$

Let $\delta_t = r_{t+1}+\gamma\cdot\max_{a\in\mathcal{A}} Q(s_{t+1},a;\theta') - Q(s_t,a_t;\theta)$ be the Temporal Difference (TD) error. From Eq.11 it is clear that we aim to minimize the magnitude of the squared TD error. Hence, this method picks the samples with the largest error so that our neural network can minimize it. Following this method, one shall insert the TD error into the replay buffer to assign priorities $|\delta_i|$, e.g., our replay buffer samples are $(s_t, a_t, s_{t+1}, r_t, |\delta_t|)$ (We neglect the done flag for convenience reasons). One of the main problems with this method, is how to keep track of all the $|\delta_i|$ in the replay buffer while training our neural network? In order to reduce this method complexity, computationally efficient alternative is to only update the δ_i terms for items that are actually sampled during the mini-batch gradient updates. Since we already have to compute δ_i anyway to get the loss, we might as well use those to change the priorities. For a mini-batch size of M, each gradient update will change the priorities of M samples in the replay buffer, but will not change the rest of the samples. Next, given the absolute TD errors, we get a probability distribution for sampling using the following method, let $p_i = |\delta_i| + \epsilon_n$ for some small positive ϵ_n(the small ϵ_n purpose is to make sure that every sample has some non-zero probability of being drawn). During exploration, the p_i terms are not known for brand-new transitions because those have not been evaluated with the networks to get a TD error term. To get around this, PER initializes p_i according to the maximum priority of any priority thus far, thus favoring those terms during sampling later. Now, one can easily obtain a probability distribution over the samples in the replay buffer,

$$\mathbb{P}(i) = \frac{p_i^\alpha}{\sum_j p_j^\alpha} \quad (12)$$

Where α determines the level of prioritization. For instance, when $\alpha \to 0$ we get no prioritization and respectively, when $\alpha \to 1$ we get "full" prioritization. While we term "full" prioritization as the case that our sampling data points are more heavily dependent on the actual δ_i values. Now, after we've defined our probability distribution $\mathbb{P}$ over each sample in the replay buffer, as a last step, we need to explain the importance sampling phase, which aims to overcome the induced bias of the prioritization between different samples. In order to apply

importance sampling, we use the following weights,

$$w_i = (\frac{1}{N} \cdot \frac{1}{\mathbb{P}(i)})^{\beta} \tag{13}$$

After that, we further scale the weights in each mini-batch such that $\max_i w_i = 1$ for stability reasons. In addition, the $\frac{1}{N}$ term is because of the current experience replay size (how many samples we currently have in the replay buffer). The β term controls how much prioritization to apply. The authors argue that training is highly unstable at the beginning, and that importance sampling corrections matter more near the end of training. Thus, β starts small (values of 0.4 to 0.6 are commonly used) and anneals towards one. Finally, we update the weights using the following update rule,

$$\theta \leftarrow \theta + w \cdot \delta \cdot \nabla_{\theta} Q \tag{14}$$

3.4 Experiments & Results

First, we added the reward-shaping mechanism to the training procedure. The results, presented in Fig.14, show that this method indeed improves the algorithm's training stability and its performance. Although using only RS_t^1 leads our agent towards a slower convergence. We claim that this method improves the training stability since it increase the average score curve smoothness. Thus, from here on we will examine the results while considering reward shaping. In Fig.15 we present the avg score of the following combinations of improve-

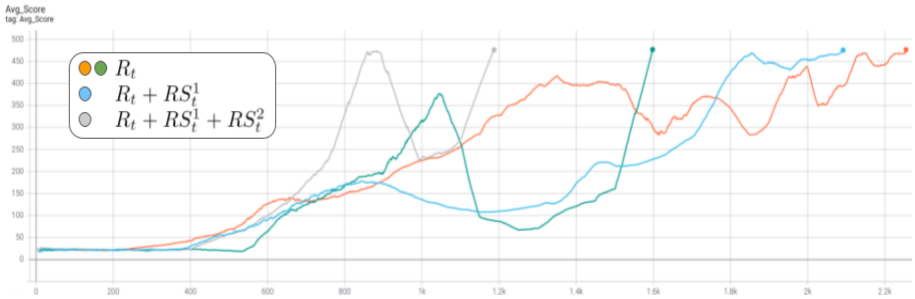


Figure 14: Avg-Score with different reward shaping methods

ments: DQN with full reward-shaping, DDQN with full reward-shaping, DQN with PER and full/partial reward-shaping, DDQN with PER and full/partial reward-shaping. From Fig.15, we claim that the best model was DQN with PER and partial reward-shaping (red). The reason this combination of methods performed significantly better than DQN with PER and full reward-shaping (pink), in our opinion, is that the sparse reward RS_t^2 can cause high priority in the PER buffer for learning the 'failing states' and it doesn't focus on the 'good states' sufficiently. We can also observe similar phenomena when looking at the graphs of DDQN and PER combined with different reward-shaping (green

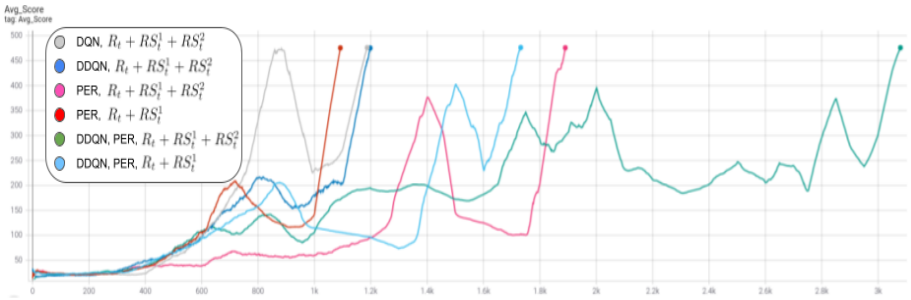
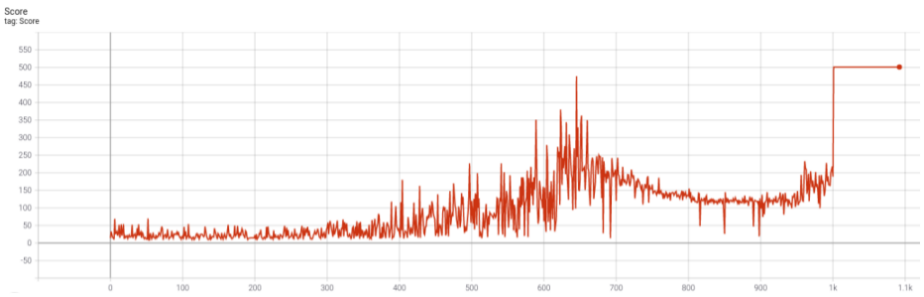


Figure 15: Avg-Score with different combinations of 'improvements'

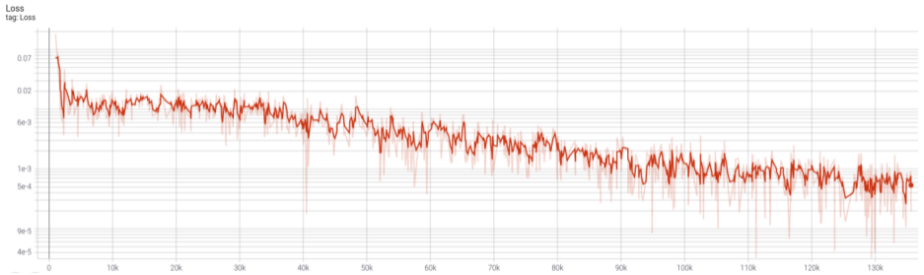
vs cyan). We can also observe from the results that DDQN with full reward-shaping is more stable than DQN with the same reward-shaping. This result is reasonable because the goal of DDQN method is to combat the maximization bias of the traditional DQN method. Comparing the usage of PER with DQN and DDQN, we notice a strange degradation in performance in both reward-shaping methods. In order to explain this phenomena we need to recall that the PER is using the estimated TD-error δ_t in order to determine priority of each transition, while in DDQN we modify the TD-error estimation according to $\tilde{\delta}_t = Y_t^{DoubleDQN} - Q(S_t, A_t; \theta_t)$, therefore the priority changes in a different manner. It is possible that if in our implementation we would use the same δ_t as in DQN, the results would be better.

3.5 Results

In Fig.16 we present the score over episodes, and loss over time-steps of our finest model - DQN with PER and partial reward-shaping. The hyperparameters used are shown in Table 1 (3-layers network). We can observe that the agent reaches a score of 500 since episode 1001 consecutively until episode 1092 when he reaches an avg score of 475.6.



Score



Loss

Figure 16: Score and Loss of the best model