SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF COMPUTER, CONTROL, AND MANAGEMENT ENGINEERING ANTONIO RUBERTI (DIAG)

Assignment 3

REINFORCEMENT LEARNING

Professors:
Roberto Capobianco

Students:
Luca Del Signore
(2096442)

Academic Year 2024/2025

Contents

1 Exercise 1 (theory)

In order to calculate w_0, w_1 and θ after the transition $(s_0, a_0 = 1, r_1 = 1, s_1, a_1 = 0)$ we need to apply the following steps:

Algorithm 1 1-Step Actor-Critic Update with LFA

```
1: $\delta \leftarrow r_1 + \gamma Q_w(s_1, a_1) - Q_w(s_0, a_0)$
2: $w \leftarrow w + \alpha_w \delta * \nabla_w Q_w(s_0, a_0)$
3: $\theta \leftarrow \theta + \alpha_\theta \delta * \nabla_\theta \ln \pi_\theta(a_0 = 1|s_0)$
```

The TD-error is obtained as:

$$\begin{aligned}\delta &= 1 + 0.8 * w_0^T x(s_1) - w_1^T x(s_0) \\ &= 1 + 0.8 * (0.5, 0) \begin{bmatrix}1\\0\end{bmatrix} - (1, 0) \begin{bmatrix}0\\1\end{bmatrix} \\ &= 1 + 0.8 * (0.5) - 0 \\ &= 1.4\end{aligned}$$

For computing the update of w_1, we can directly assert that:
$\frac{\partial}{\partial w_1} Q_w(s_0, a_0) = \frac{\partial}{\partial w_1}[w_1 * x(s_0)] = x(s_0)$. Substituting this result into the second step of 0 we obtain:

$$\begin{aligned}w_1 &= w_1 + 0.5 * (1.4) * \begin{bmatrix}0\\1\end{bmatrix} \\ &= \begin{bmatrix}1\\0\end{bmatrix} + \begin{bmatrix}0\\0.7\end{bmatrix} \\ &= \begin{bmatrix}1\\0.7\end{bmatrix}\end{aligned}$$

For computing the update of w_0, we need to consider that a=0 was not chosen at time step 0. Thus $\frac{\partial}{\partial w_0} Q_w(s_0, a_0 = 1) = \frac{\partial}{\partial w_0}[w_1 * x(s_0)] = 0$, and then w_0 is not changing.

For the final part of the update step, we also need to consider θ, which are the parameters of the policy. The policy is defined as $\pi_\theta(a = 1|s) = \sigma(\theta^T x(s))$ (where $\sigma()$ is the sigmoid function), representing the probability distribution of choosing $a = 1$ given s. Referring to the third step of 0 we need to compute $\nabla_\theta \ln \sigma(\theta^T x(s))$.
We recall that:
$\frac{d}{dz} \ln(f(z)) = \frac{1}{f(z)} \cdot \frac{d}{dz} f(z)$ and that $\frac{d}{dz}\sigma(f(z)) = \sigma(f(z)) \cdot \big(1 - \sigma(f(z))\big) \cdot \frac{d}{dz} f(z)$.

Applying the chain rule, we have:

$$\nabla_\theta\big(\ln\sigma(\theta^T x(s))\big) = \frac{1}{\sigma(\theta^T x(s))} * \nabla_\theta\big(\sigma(\theta^T x(s))\big)$$

$$= \frac{\cancel{\sigma(\theta^T x(s))}}{\cancel{\sigma(\theta^T x(s))}}(1 - \sigma(\theta^T x(s)) * \nabla_\theta\big(\theta^T x(s)\big)$$

$$= (1 - \sigma(\theta^T x(s))) * x(s)$$

By substituting the numerical values:

$$\begin{aligned}
\theta &= \theta + \alpha_\theta \delta * (1 - \sigma(\theta^T x(s_0)) * x(s_0) \\
&= \begin{bmatrix}0.6\\1\end{bmatrix} + 0.5 * 1.4 * \big(1 - \sigma(\begin{bmatrix}0.6 & 1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix})\big) * \begin{bmatrix}0\\1\end{bmatrix} \\
&= \begin{bmatrix}0.6\\1\end{bmatrix} + 0.5 * 1.4 * \big(1 - \sigma(1)\big) * \begin{bmatrix}0\\1\end{bmatrix} \\
&= \begin{bmatrix}0.6\\1\end{bmatrix} + 0.5 * 1.4 * \big(1 - 0.731\big) * \begin{bmatrix}0\\1\end{bmatrix} \\
&= \begin{bmatrix}0.6\\1\end{bmatrix} + 0.5 * 1.4 * \big(0.269\big) * \begin{bmatrix}0\\1\end{bmatrix} \\
&= \begin{bmatrix}0.6\\1\end{bmatrix} + \begin{bmatrix}0\\0.1883\end{bmatrix} \\
&= \begin{bmatrix}0.6\\1.1883\end{bmatrix}
\end{aligned}$$

2 Exercise 2 (practical)

2.1 Overview

In this chapter we are going to see in detail the implementation of the DDQN [2] algorithm with Prioritized Experience Replay (PER) Buffer [5] in the discrete gym environment "CarRacing-v2" which provides images as observations. The architecture of the neural networks used in the DDQN pipeline are the same as those employed by DeepMind for Atari games [4] proving to be flexible and adaptable across different games. The code is organized following the software structure adopted for the practical 7 of the course [1]

2.2 ZIP structure

Before proceeding, I will anticipate the structure of the zip file and explain the purpose of its contents. Throughout this work, we are going to analyze in depth all the most important functions contained in the Python files. The "car racing" folder includes the following files:

- `model_fast.pt` is the fastest model I have found, and by default it will be loaded;
- `model_precise.pt` is the most precise model I have found;
- `student.py` contains all the functions used during training and also utilities functions for preprocessing images and storing data;
- `other_implementation.py` is similar to `student.py` but instead has a different implementation of the PER buffer using the SumTree data-structure ([3]).

2.3 Pre-Processing

Since the DeepMind network expects as input a stack of four 84x84 grayscale frames, some preprocessing steps are necessary as the environment provides a single rgb image of 96x96 pixels after each action. The preprocessing, contained in the `preprocess` function, involves:

1. Converting the image to grayscale;
2. Cropping the image in order to be 84x84 eliminating useless part (for example the bars below);
3. Normalizing the image between 0 and 1 dividing each pixel by 255;

4. Adding this image as the first in the stack of the last four images and removing the oldest one. The resulting stack of four frames now represents the next state in my pipeline. This approach expands the state representation using four consecutive frames.

Additionally, during training, it is necessary to skip the first 60 frames at the beginning of each episode giving no action to the Agent. After that, frames are stacked in groups of four to create the input expected by the network. This logic is contained in the function `super_restart`.

2.4 The spin-off turn problem

By "spin-off turn problem" I refer to the situation where the car approaches a hairpin turn at high speed, resulting in a skid while attempting to follow the turn, ending up in a 180-degree spin away from the correct driving direction. This is the most challenging situation that can be encountered during the simulations, as most of the models that I have tested either concluded the simulation in the grass or continued the lap in the wrong way. However, with fine-tuning of prioritization hyperparameters, some models managed to recover from this type of "incident" and resume driving on the correct direction in the track. I also observed a trade-off between speed and precision in these critical situations.

2.5 Reward Reshaping

The `take_step` function contains the logic for assigning rewards during training. When called in "explore" mode, it performs a random action; otherwise, it selects the best action found so far ("exploit" mode). After taking a step, the environment returns the consequence of that action through a single RGB image, the reward assigned by the environment and a flag indicating whether the car has explored all tiles on the track or reached the maximum number of frames. The newly obtained image is appended to `self.last_frames` which contains the last 4 frames. This stack of frames will be sent to the network enabling it to infer the car's movement. Additionally, if the car stops receiving positive rewards for 100 consecutive steps (that is set in `self.max_patience`) it will gain -100 as reward simulating a "death" scenario. This penalizes the car for wasting time in exploring tiles, whether by moving too slowly or staying on the grass for too long. This logic is managed by the boolean variable `patience_truncation`. Instead, when the car explores all the tiles or reaches the limit of frames, the variable `done` is set to `True`. Each new transition is stored in the PER buffer with a priority equal to the maximum priority among all samples already present in the buffer. It is important to note that the environment will set the boolean variable `truncated` to `True` also when the the car visits all the tiles rather than using `terminated`. To handle this, both flags are saved in the buffer. This approach ensures that if the car completes all frames without "dying" (thus avoiding the max-patience trigger), the final transition is stored as a stable transition. When either `max_patience` is reached or `done` becomes `True`, the environment is reset using the `super_restart` function described earlier. Moreover, during training, I reduced the maximum episode steps to 930 from 1,000 because I noticed that the car was sometimes moving too quickly and completing the lap missing some tiles. As a result, the lap would finish without triggering the `done` flag, but the "max patience" condition would be triggered instead, since the car had already explored the initial tiles early on in the lap, leading to no positive reward in the later stages. This caused the agent to receive a reward of -100 at the end simply because the "max patience" limit had been reached, even though the car had missed only a few tiles and arrived at the end of the track. To avoid this issue, the maximum episode steps were reduced so that the lap can be considered completed even if it was truncated just before the end. If the agent was able to reach the end of the 930 frames without triggering the max patience, it would likely have completed the remaining 70 frames without problems.

2.6 Prioritized Experience Replay Buffer

In this section we will dive into the implementation and theory behind the Prioritized Experience Replay (PER) buffer. The primary goal of PER is to bring up again transitions to the network that occur rarely and hold greater learning potential, as opposed to redundant situations. Critical transitions (rare successes) are often hidden within a mass of highly repetitive failure cases. Moreover, some transitions may not be immediately useful to the agent, but might become valuable as the agent competence increases; this approach leads to a policy that is robust even in rare scenarios.
To achieve this, a prioritization system of replay is necessary, in contrast to the random replay of transitions in traditional Experience Replay Buffers. I have chosen as prioritization system the Temporal Difference error (TD-error) associated with each transition because it indicates how surprising or unexpected that transition was. This is especially effective in deterministic domains with no noise.
However, this type of prioritization introduces some issues:

1. **Loss of Diversity** as it ignores completely samples with low TD-error;
2. **Bias** as the network will see samples that do not reflect a real distribution of experiences.

In order to mitigate the Loss of Diversity problem we need to use Stochastic proportional prioritization that interpolates between pure greedy prioritization and uniform random sampling based on how much the α term is set (larger α increases the emphasis on prioritization). The probability of sampling the i-th transition from the buffer is given by:

$$P(i) = \frac{p_i^\alpha}{\sum_n p_n^\alpha} \tag{1}$$

where n are the total number of samples in the buffer and $p_i = |\delta_i| + \epsilon$ is TD-error of that transition plus an epsilon that ensures a small chance of revisiting a transition even if its TD-error is zero. When $\alpha = 0$, the distribution becomes uniform for each sample, equivalent to normal Experience Replay Buffer.

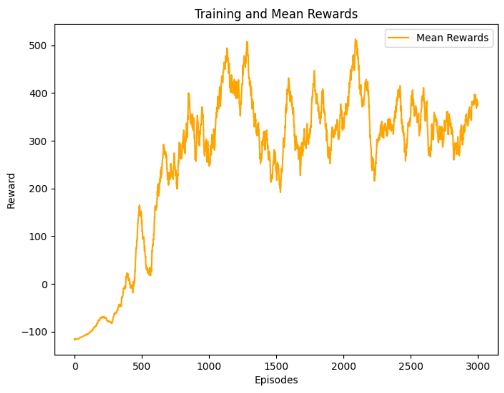
For addressing the bias problem, we can correct it by using importance-sampling (IS) weights, which are applied to the TD-errors of the sampled transitions during the update phase of the Q-Learning algorithm:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta \tag{2}$$

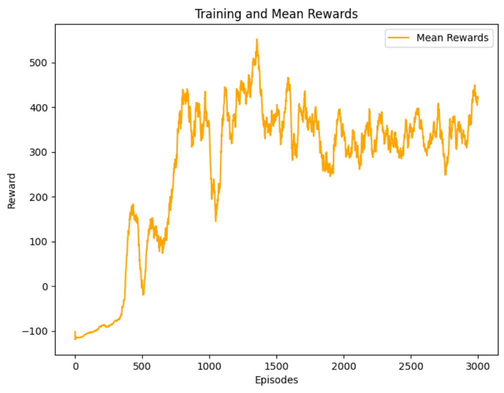
These weights reduce the gradient variation for samples with low weights (and then High probability of being sampled). The larger β is, the more the correction is applied to the distribution. For stability reason, the weights are normalized each step by the maximum value of weights found among the batched samples.

There are several ways to implement a buffer that manages priorities. The vanilla and simplest implementation that I have used in `student.py` stores both the transitions and their p_i^α values in parallel arrays, which are updated only when transitions are re-proposed. However, since we need to compute $\sum_n p_n^\alpha$ at each update step, the array that stores p_i^α can be more efficiently implemented using a SumTree, where the parent node contains the sum of its children. I have integrated the SumTree implementation from [3] in `other_implementation.py` for comparison with the vanilla approach.

As shown in Figure 1, the performance in terms of mean rewards (using a 50-episode window with $n = 50,000$ samples stored) is similar for both implementations and reaches the same peak. The main adavtage of using the SumTree is that in the root there is already the sum of priorities, calculated in $O(k \log n)$ steps (where k is the batch size) during the update priorities step. In contrast, our vanilla PER buffer requires to recompute $\sum_n p_n^\alpha$ each time that takes $O(n)$ steps. Thus, with equal performance, the SumTree implementation is faster and uses less RAM, enabling us to store more samples, and as we will see in the chapter 2.8 this is crucial for having a more robust policy. When a new sample is added to the PER buffer, it is assigned with the maximum priority found in the buffer in order to guarantee to be sampled at least once during the update step. At the beginning of the train, during the burn-in phase when we populate the buffer with initial samples, the priority is set to ϵ. After the re-evaluation, the priority will become $p_i^\alpha = (|\delta_i| + \epsilon)^\alpha$. When the buffer reaches its capacity, it starts overwriting from the beginning in a circular fashion.



(a) Vanilla PER



(b) SumTree PER

Figure 1: Mean rewards with Buffer Size of 50,000 samples

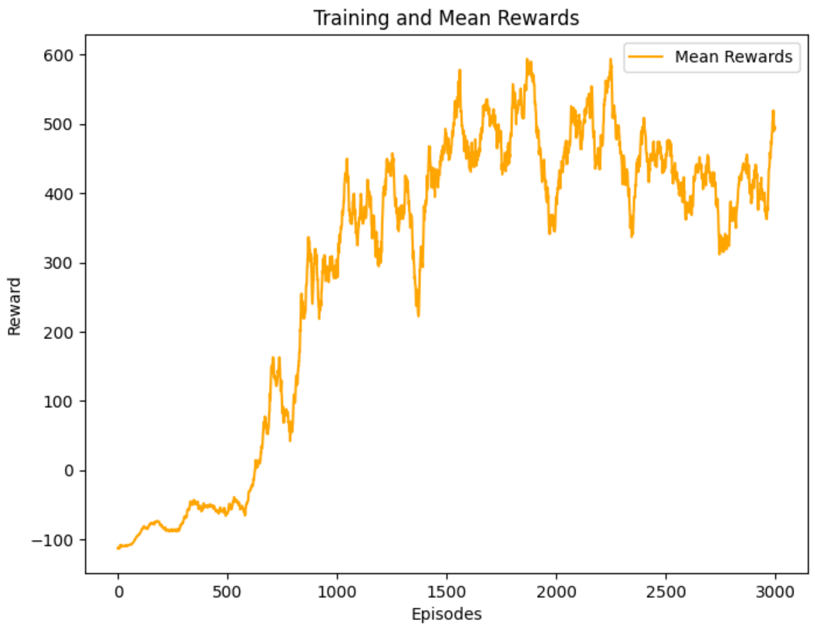


Figure 2: Mean rewards with Buffer Size of 100,000 samples

2.7 DDQN and training process

The principle behind Double Deep Q-learning is to use two neural networks as value function approximators that take as input 4 stacked consecutive images and return the Q-value for each action. One is used to select the next action (the normal one Q) and the other to evaluate the action (the target one Q_{target}). However, only the normal network is modified during the update step, while the target network remains frozen and is updated periodically every 100 steps by copying the weights of the normal network. This process aims to reduce bias and improve overall stability. In the training process, which is handled by the `train` function, initially we have a burn-in step in which the buffer is populated with 20,000 steps of random actions for full exploration. After this, the actual train begins: The agent commits 10 ϵ-greedy steps (different from the value of ϵ used for priority) that are saved in the buffer, and then it will do the update phase of the Q normal network in which it will sample a batch of k (in our case $k = 64$) transitions from the buffer based on the probability described in 1 and then, these samples, will be used to compute the TD-error:

$$\delta_j = R_j + \gamma Q_{\text{target}}\big(S_j, \arg\max_a Q(S_j, a)\big) - Q(S_{j-1}, A_{j-1}) \tag{3}$$

To train the network using IS mechanism, each pair sample-TD_error is multiplied by its corresponding weight. The loss function is then computed as the mean of the squared weighted TD-errors among all the batched samples:

$$\texttt{loss = torch.mean((TD\_errors)}^2 \cdot \texttt{weights)}\,. \tag{4}$$

The TD-error is squared because, when differentiating it with respect to the parameters of the normal network, we obtain something similar to the pseudocode presented in [5] for DDQN+PER:

$$\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1}) \tag{5}$$

This logic is implemented in `calculate_loss` function.

2.8 Results and comparisons

In this section I will present the experiments conducted and the considerations derived from them. After several trials, It became clear that starting with a burn-in phase and setting the initial $\epsilon = 0.7$ for ϵ-greedy (with a decay of 0.99 after an episode) was more effective than starting with $\epsilon = 1$ without the burn-in phase. Moreover, copying in the target network Q_{target} the weights of the normal network Q every 5,000 steps caused very slowly convergence. So I set up it to 100 steps. However, the most interesting findings emerged from tuning the α and β priorization parameters.
Setting $\alpha = 0.1$ and $\beta = 0.1$, I have observed visually that the agent performed well in general, but it struggled in some rare and critic situations on which it has not learned

very well. For example in the spin-off turn problem 2.4 it failed to recover. Another important error was that sometimes it turned in the opposite way. It is not surprising as little values for α and β make the system behave similarly to a random sampling buffer, skipping some important and rare situations hidden the ocean of samples. Thus, in order to prioritize more and more this kind of rare situations, I have used the parameters indicated in the [5] for DDQN: it suggests an $\alpha = 0.6$ and a β that starts from 0.4 and gradually increasing to 1 by the end of training. These parameters' settings highlighted more rare and critic situations in contrast to the common and cruise situations. Visually, now the car rarely goes out in the grass or makes rough errors, and consequently it reached better results. Even in the spin-off turn problem 2.4 it now manages to recover in the right direction most of the time. With these tuned values of α and β, I explored how buffer capacity affects agent performance. Starting with a capacity of 50,000 samples 1 and then increasing it 100,000 samples 2, the results confirmed that larger memory buffers improve performance, leading to higher rewards:

$\alpha = 0.6, \beta = 0.4 \rightarrow 1$	**Buffer capacity=50,000**	**Buffer capacity=100,000**
Highest Mean Reward	520	592
Cruise Mean Reward	300-400	400-500

Table 1: Comparisons between different values of Buffer size.

It's singular to note in 2 that the mean reward reached a peak in the 1900-th episode and then it started to decrease instead of stabilizing at that level. In order to understand what was happening, I have saved both the model at the end of the 1900-th episode when the mean reward was at its peak and at the end of the training (that are respectively `model_fast.pt` and `model_precise.pt`). I have seen that both models performed well with no major mistakes, but the real difference relies in the lap completion speed: the first is faster, generating the problem described in 2.4 that it now manages to recover in the right direction most of the times; instead, the last model, is slower but never (or almost never) generates problems in bends like that explained in 2.4. However, due to the environment's time constraints in terms of frames, the slower model had less time to accumulate rewards, explaining the reward drop after the 1900-th episode. In other words, the faster model seems to be more courageous, taking aggressive actions that complete the lap quickly but risking occasional recovery failures. On the other hand, the slower model seems to be more "timid" and cautious, focusing on precision and minimizing mistakes.

2.9 Future Works

As increasing the buffer capacity led to better results, it suggests that some samples should remain in the buffer longer before being replaced (for example when the buffer reaches full capacity). So it would be interesting in investigating a system of decision for which sample to override and which sample to let in the buffer for longer time.

3 Collaborators

I have collaborated with Massimo Romano (2043836) for this Assignment.

References

[1] KRL Group. *DDQN Solution Notebook*. Accessed: 2024-12-07. 2024. URL: https://github.com/KRLGroup/RL_2024/blob/main/p06_DDQN/DDQN_sol.ipynb.

[2] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning*. 2015. arXiv: 1509.06461 [cs.LG]. URL: https://arxiv.org/abs/1509.06461.

[3] Howuhh. *Prioritized Experience Replay*. Accessed: 2024-12-07. 2024. URL: https://github.com/Howuhh/prioritized_experience_replay.

[4] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG]. URL: https://arxiv.org/abs/1312.5602.

[5] Tom Schaul et al. *Prioritized Experience Replay*. 2016. arXiv: 1511.05952 [cs.LG]. URL: https://arxiv.org/abs/1511.05952.