University of Liège

Assignment 3

Optimal decision making for complex problems

Maxime Meurisse (s161278)
François Rozet (s161024)

Academic year 2020-2021

1 Domain

From the previous assignment, we keep our implementation of the different components of the *car-on-the-hill* problem, as well as the visual rendering tools, *i.e.* the `domain.py` and `display.py` files, respectively. However, by simplicity, we modify the action space to $U = \{0, 1\}$ and update the environment dynamics accordingly, *i.e.*

$$\dot{s} = \left(\frac{8u - 4}{m} - gHill'(p) - s^2 Hill'(p)Hill''(p)\right) \frac{1}{1 + Hill'(p)^2}.$$

Furthermore, in this assignment, the agent has no longer access to the state $x \in X$ of the environment. Instead, it has access to a (visual) observation $v(x)$ of the state x; in this case an image rendered by the "graphics engine" of the game.

In practice, the observation is produced by the `state2visual` function, which retrieves a 400×400 RGB image from the engine and down-samples it to 100×100 pixels, to reduce memory consumption. The color spaces are also mapped from $[0; 255]$ to $[0; 1]$.

It should be noted that, in a general setting, a single observation $v(x)$ is not sufficient for an agent to understand the state x. For example, in a video game like Super Mario, it would not be possible to estimate the movement speed or direction of objects from a single screenshot. Hence, several successive observations are usually given to the agent. However, in the case of the car-on-the-hill problem, this is not necessary as a speedometer is provided on the interface.

2 Deep Q-Learning

In this section, we apply the *online Q-learning* algorithm in order to infer the best strategy for our agent. As required by the statement, we use a (deep) neural network architecture as an estimator $\hat{Q}$ for the Q-function, which is usually referred to as *Deep Q-learning* (DQL) or *Deep Reinforcement Learning*.

Note. All the routines mentioned in the rest of this document have been implemented in the `dql.py` file.

2.1 Architecture

Our network architecture takes inspiration in image classification networks, like AlexNet [1] or VGG [2]. It is composed of two successive parts: a fully convolutional network (FCN), to extract features, and a dense multi-layer perceptron (MLP), to process those features.

Like Mnih et al. [3], in order to amortize the cost of the forward pass and because the number of possible actions is finite, we use an architecture in which there is a separate output value for each action, and only the state observation is given as input to the network.

$$\hat{Q}(v(x)) \approx \begin{pmatrix} Q(x, 0) \\ Q(x, 1) \end{pmatrix} \tag{1}$$

More specifically, the FCN part of our network contains four groups of layers, each composed of a *Double Convolution* (two consecutive convolution layers), a batch normalization

layer and a ReLU activation function. We made the choice to use double convolutions based on previous computer vision projects we have realized in the past (e.g. [4] and our respective internships at EVS) as well as associated scientific literature [5, 6]. The MLP part, on the other hand, is very similar to the one used in previous assignment : 3 hidden layers of 8 neurons, along with input and output layers, with ReLU activation functions.

The overall architecture is described in Table 1.

	Layer	**Neurons**	**Kernel**	**Stride**	**Padding**	**Output**
	Input	-	-	-	-	(3, 100, 100)
FCN	Conv1	16	5	2	2	(16, 50, 50)
	Conv2	32	5	2	2	(32, 25, 25)
	Conv3	64	5	2	2	(64, 13, 13)
	Conv4	128	5	2	2	(128, 7, 7)
	Flatten	-	-	-	-	(6272)
MLP	Dense1	8	-	-	-	(8)
	Dense2	8	-	-	-	(8)
	Dense3	8	-	-	-	(8)
	Dense4	8	-	-	-	(8)
	Dense5	2	-	-	-	(2)

Table 1 – Architecture of the network used for the DQL algorithm.

Note. This network architecture and its training routine were implemented using PyTorch.

2.2 Routine

We also take inspiration in Mnih et al. [3] for our training routine. The agent/network is trained over 100 epochs of 25 episodes. Each episode simulates a trajectory of the car-on-the-hill environment for which the agent selects and executes actions according to an ϵ-greedy policy. However, we choose to reduce (exponential decay) the probability ϵ as the agent gains experience.

$$\epsilon = 0.05 + 0.9 \exp\left(-\frac{epoch}{25}\right) \tag{2}$$

This allows the agent to initially explore a lot of the state-action space and, then, gradually refining the knowledge of the agent at the states it would most likely visit.

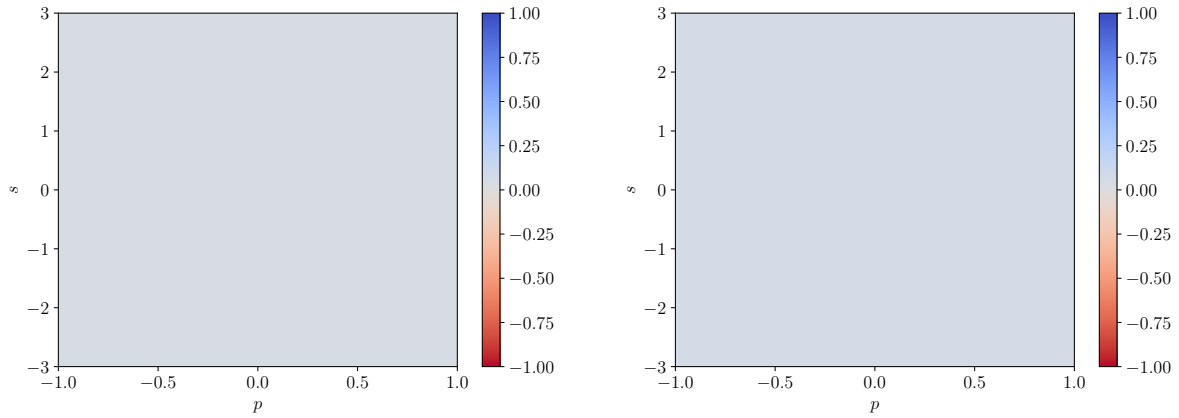
Additionally, we implement *experience replay* [7] were we store the agent's transitions $(v(x), u, r, v(x'))$ in a *replay memory* $\mathcal{D}$, implemented as a cyclic buffer of fixed capacity (4096).

After each transition, a mini-batch of (128) transitions is drawn at random from $\mathcal{D}$ with which we apply a Q-learning-like update using the Adam optimizer [8]. This method has several advantages over standard Q-learning [3, 9]. First, since the transitions are used more than once, the method usually requires less calls to the environment, which could save a lot of computation for expensive simulations. Second, learning from consecutive

transitions, which are strongly correlated, is inefficient. Conversely, sampling randomly from $\mathcal{D}$ de-correlates the samples of the mini-batch. This is well summarized by the Algorithm 1 of Mnih et al. [3].

2.3 Results

After training, we use the network to estimate the state-action values $Q(x, u)$ on a uniformly spaced (resolution of 0.01) grid of the state space for both possible actions. In Figure 1 and in the following, the state-action values are displayed on separate figures for each action.



(a) $\hat{Q}(v(x), 0)$ (b) $\hat{Q}(v(x), 1)$

Figure 1 – $\hat{Q}$ values predicted by DQL.

We observe that the network predicts a null state-action value everywhere. This clearly means that it is unable to grasp the environment dynamics and, most importantly, the reward system.

As one could expect, the expected return[1] $J^{\hat{\mu}^*_N}$ of the policy inferred from $\hat{Q}$

$$\hat{\mu}^*(x) \in \arg\max_{u \in U} \hat{Q}(v(x), u) \tag{3}$$

is null as well.

We repeated this experiment several times while modifying some hyper-parameters and the architecture, but it never worked out. Actually, this is not very surprising as it was mentioned during the course [9] that it is very hard to perform regression on moving targets, which DQL implements. In fact, this is the main reason why a target network was introduced in Mnih et al. [10]. This will be covered in Section 4.

3 Deep Q-Learning VS FQI-Trees

In this section, we compare our Deep Q-learning algorithm working with images as inputs with the algorithms implemented for the previous assignment.

[1] The expected return was estimated using the routines implemented for the previous assignment. The complete procedure is explained in the Appendix A.

3.1 FQI with XRT on images

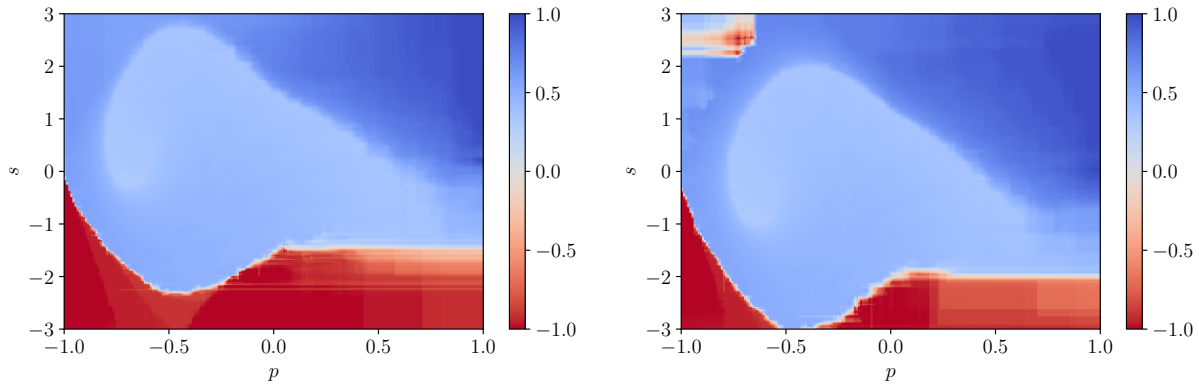
Unfortunately, we were not able to implement Fitted-Q-Iteration (FQI) with Extremely Randomized Trees (XRT) for $v(x)$ inputs due to technical constraints. Indeed, (extremely) randomized trees is an offline estimator meaning that it requires to have the entire dataset stored in memory at once. Additionally, the number of nodes in randomized trees grows (linearly) with the number of estimators, the number of samples and (linearly) with the number of input features. Therefore, because the observations are 100×100 RGB images, we didn't manage to realize this experiment on our personal computers.

Also, while searching for solutions, we noticed that there is a lack of literature around the topic of Randomized Trees applied to images. In fact, we were only able to find a few (rather old) relevant publications: Ernst et al. [11], Bosch et al. [12], Marée et al. [13]. This could indicate that such methods are not appropriate for this task, and therefore would have performed poorly. Although, it is very unlikely that it would have performed worse than the network of Section 2.

3.2 FQI with XRT and Parametric Q-learning with direct access to x

Conversely to the previous section, here, the estimators have direct access to the state x. In fact, this corresponds to the experiments we realized in the previous assignment. Therefore, instead of re-implementing the experiments, we simply re-use the results and plots of the previous report.

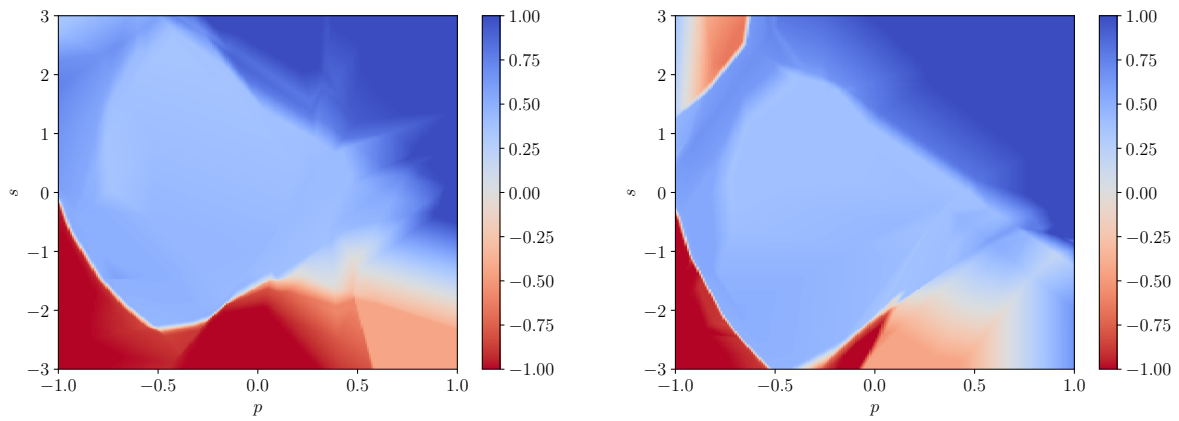
It should be noted that we only kept the results with the "Monte Carlo" dataset generation strategy, *i.e.* joining trajectories. Also, for FQI with XRT we selected the first stopping rule, *i.e.* stop when $N = 162$. Finally, for the parametric Q-learning (PQL), we used the Adam optimizer instead of regular SGD as the latter didn't produce relevant results. The results are presented in Figures 2 and 3.



(a) $\hat{Q}_N(x, 0)$ (b) $\hat{Q}_N(x, 1)$

Figure 2 – $\hat{Q}_N$ values predicted by FQI with XRT; $N = 162$.

Concerning the expected returns of the inferred policies, FQI with XRT reached 0.42 while PQL reached 0.37. These scores demonstrate a good understanding of the environment dynamics and reward system. But, it is not surprising that these methods perform much better than DQL on observations, as they have access to the "true" states. In Section



(a) $\hat{Q}(x, 0)$ (b) $\hat{Q}(x, 1)$

Figure 3 – $\hat{Q}$ values predicted by PQL.

2, the network not only has to understand that its actions have an effect on the position and speed of the car, but also that there is a car and that it moves. Hence, the problem is inherently harder.

However, the problem could have been even harder if the provided observations were too noisy or not enough informative to derive with certainty the internal state of the environment. This is the problem of *acting under uncertainty* [14].

4 Deep Q-Learning VS Deep Q-Network

In this section, we compare our implementation of the *Deep Q-learning* algorithm with the *Double Q-learning* [9, 15] algorithm popularized by Mnih et al. [10].

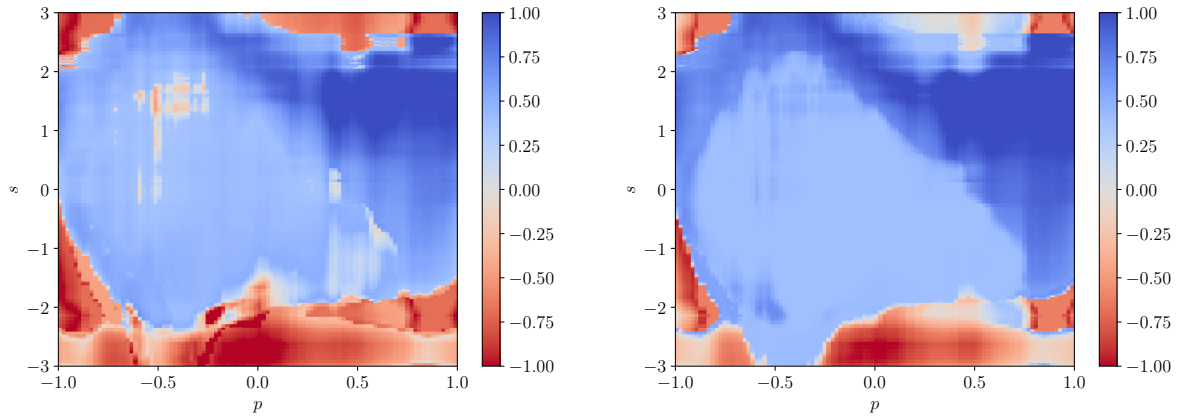
Double Q-learning is exactly the same as Deep Q-learning, with the exception that it introduces a second neural network, called the *target network*, that replaces the first network when computing the targets in the Q-learning updates. In practice, this target network is a clone of the main network and is updated periodically with the new weights of the latter. In their paper, Mnih et al. [10] explain that

> "This modification makes the algorithm more stable compared to standard online Q-learning, where an update that increases $\hat{Q}(x, u)$ often also increases $\hat{Q}(x', u')$ for all u' and hence also increases the target $r + \gamma \max_{u'} \hat{Q}(x', u')$, possibly leading to oscillations or divergence of the policy. Generating the targets using an older set of parameters adds a delay between the time an update to $\hat{Q}$ is made and the time the update affects the targets, making divergence of oscillations much more unlikely."

For our implementation of Double Q-learning, we keep the same network architecture and the same training routine as in Section 2, but we clone the main network at the beginning of each epoch to produce a target network.

4.1 Results

The state-action values, for each action, obtained using Double Q-Learning are shown in Figure 4.



(a) $\hat{Q}(v(x), 0)$ (b) $\hat{Q}(v(x), 1)$

Figure 4 – Q-values predicted by Double-QL.

This time, the network seems to have understood fairly well the environment dynamics as the values of $\hat{Q}$ are quite close to those of FQI with XRT or PQL (*cf.* Figures 2 and 3). We do note, however, that the predictions are less "smooth" than those of FQI (or PQL) and present steep variations in some locations. Most importantly, Double Q-learning did a much better job than standard Deep Q-learning. Especially, the expected reward of the inferred policy reaches 0.38, *i.e.* barely worse than FQI (with XRT) and a bit better than PQL, while having only access to observations $v(x)$.

A Expected Return of a Policy in Continuous Domain

Note. This section is a copy of the corresponding section in our previous assignment.

In the car-on-the-hill problem, the dynamics f and reward signal r of the environment are deterministic, *i.e.* a state-action pair (x, u) will always lead to the same one-step transition (x, u, r, x'). However, the choice of the initial state x_0 is stochastic. Therefore, the expected return J^μ of a policy μ can be expressed as

$$J^\mu = E_{x_0 \sim p(x_0)} \left\{ \lim_{N \to \infty} \sum_{i=0}^{N-1} \gamma^i r(x_i, \mu(x_i)) \right\} \tag{4}$$

where $x_{i+1} = f(x_i, \mu(x_i))$ and $\gamma \in [0; 1)$ is the discount factor. Interestingly, because r is non-null only once in a trajectory, we have

$$\|J^\mu\|_\infty \leq \|r\|_\infty = B_r = 1. \tag{5}$$

In order to estimate the expected return, we have to fix a time horizon N such that

$$J^\mu_N = E_{x_0 \sim p(x_0)} \left\{ \sum_{i=0}^{N-1} \gamma^i r(x_i, \mu(x_i)) \right\} \tag{6}$$

is a good approximation of J^μ. In the car-on-the-hill environment, it can be shown [16, 17] that

$$\|J^\mu - J^\mu_N\|_\infty \leq \gamma^N \|J^\mu\| \leq \gamma^N B_r \tag{7}$$

which can be used to enforce a certain precision. With a threshold $\epsilon = 10^{-2}$, we have

$$\begin{aligned} \gamma^N B_r &\leq \epsilon \\ N &\geq \log_\gamma \frac{\epsilon}{B_r} \end{aligned}$$

which leads the smallest acceptable value

$$N = \left\lceil \log_\gamma \frac{\epsilon}{B_r} \right\rceil = 90. \tag{8}$$

Still, it is not possible to compute J^μ_N exactly as there is an infinite number of possible initial states x_0. Therefore, the expectation has to be approximated as well. This is done using the *Monte Carlo* method, *i.e.* approximating the expected return by the average cumulative reward over several trajectories. Let $h = (x_0, u_0, r_0, x_1, \ldots, u_{t-1}, r_{t-1}, x_t)$ denote a trajectory of length t. Since x_t can be the only terminal state of h, the cumulative reward C_h is computed as

$$C_h = \sum_{i=0}^{t-1} \gamma^i r_i = \gamma^{t-1} r_{t-1}. \tag{9}$$

Then, to estimate J^μ_N, we simulate n trajectories h_i of length N or less and compute

$$J^\mu_N \simeq \frac{1}{n} \sum_{i=0}^{n-1} C_{h_i}. \tag{10}$$

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105 (page 1).

[2] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014) (page 1).

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013) (pages 1–3).

[4] François Rozet. "Automatic Detection Of Photovoltaic Panels Through Remote Sensing". URL: https://github.com/francois-rozet/adopptrs (page 2).

[5] Long Sha, Jennifer Hobbs, Panna Felsen, Xinyu Wei, Patrick Lucey, and Sujoy Ganguly. "End-to-end camera calibration for broadcast videos". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 13627–13636 (page 2).

[6] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation". In: *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, pp. 234–241 (page 2).

[7] Long-Ji Lin. "Reinforcement learning for robots using neural networks". Tech. rep. Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993 (page 2).

[8] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014) (page 2).

[9] Sergey Levine and Damien Ernst. "Advanced algorithms for learning Q-functions." URL: http://blogs.ulg.ac.be/damien-ernst/wp-content/uploads/sites/9/2018/02/More-on-Q-Learning.pdf (pages 2, 3, 5).

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533 (pages 3, 5).

[11] D Ernst, R Marée, and L Wehenkel. "Reinforcement learning with raw image pixels as state input". In: (2006) (page 4).

[12] Anna Bosch, Andrew Zisserman, and Xavier Munoz. "Image classification using random forests and ferns". In: *2007 IEEE 11th international conference on computer vision*. Ieee. 2007, pp. 1–8 (page 4).

[13] Raphaël Marée, Pierre Geurts, and Louis Wehenkel. "Random subwindows and extremely randomized trees for image classification in cell biology". In: *BMC Cell Biology* 8.1 (2007), pp. 1–12 (page 4).

[14] R McCallum. "Reinforcement learning with selective perception and hidden state". In: (1997) (page 5).

[15] Hado Hasselt. "Double Q-learning". In: *Advances in neural information processing systems* 23 (2010), pp. 2613–2621 (page 5).

[16] Damien Ernst. "Optimal sequential decision making for complex problems". URL: http://blogs.ulg.ac.be/damien-ernst/wp-content/uploads/sites/9/2020/02/RL-1.pdf (page 7).

[17] François Rozet. "Suboptimality of stationary policies". URL: http://blogs.ulg.ac.be/damien-ernst/wp-content/uploads/sites/9/2021/02/suboptimality_bound_proof-1.pdf (page 7).