FTML practical session 9: 2024/06/14

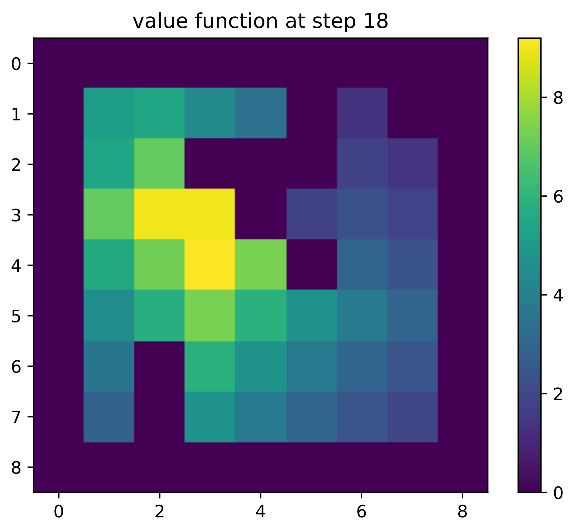


TABLE DES MATIÈRES

1 EXPLOITATION/EXPLORATION COMPROMISE

In this exercise we work with the notion of policy, applied to a simple agent in a 1-dimensional world.

1.1 Setting

We consider a one dimensional world, with 8 possible positions, as defined in the folder **project/exercise_4**. An agent lives in this world, and can perform one of 3 actions at each time step : stay at its position, move right or move left.

In this folder, you can find 3 files :

— **simulation.py** is the main file that you can run to evaluate a policy.

— **agent.py** defines the Agent class. This simple agent only has two attributes.

  — **position** : its position

  — **known_rewards** : represents the knowledge of the agent about the rewards in the worlds (see below)

— **default_policy.py** implements a default policy that consists in always going left.

Some rewards are placed in this world randomly, and are randomly updated perdiodically, at a fixed frequency. This means that a good agent should update its policy periodically as well and adapt to the new rewards. The agent knows about a reward in the world if its position has been on the same position as the reward, but each time the rewards are updated, the agents forgets all this knowledge, as implemented line 46 in **simulation.py**.

simulation.py computes the statistical amount of reward obtained by the agent and plots the evolution of this quantity in **images/**. As you can see in the **images/** folder, the average accumulated reward with the default policy is around 16, with a little bit of variance.

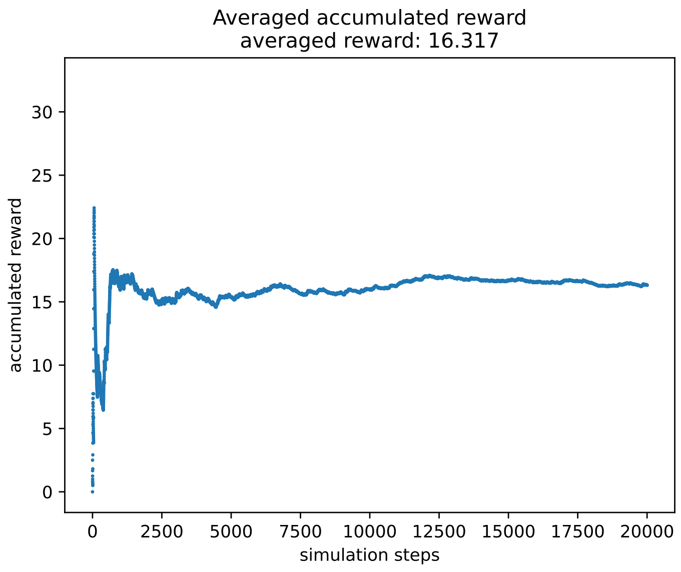


FIGURE 1 – Convergence of the average reward obtained by the agent with the default policy.

1.2 Objective

Write a different, policy better performance than the default policy.. It is possible to reach a mean score above 44. You may experiment with stochastic policies.

You will need to

— import you policy in **simulation.py**

— replace line 51 by a line that calls your policy instead of the default policy.

2 VALUE ITERATION

Implement the simplified version of the **value iteration** algorithm presented in the "lecture 8" slides.

The agent lives in a 2-dimensional world represented in Figure 2. It moves in it, as represented in figure 4, and obtains rewards, depending on its position. The agent position is perdiodically reset to the origin. The reward available is fixed, and represented in figure 3. The reward and the world configuration are contained in the **data/** folder inside **exercice_2_value_iteration/**.

You need to

— fix the **move_agent()** function

— fix the **update_value_function()** function

— You may move the agent randomly at each time step.

The estimation of the value function results from an iterative process, as displayed in figure 5 and figure 6.

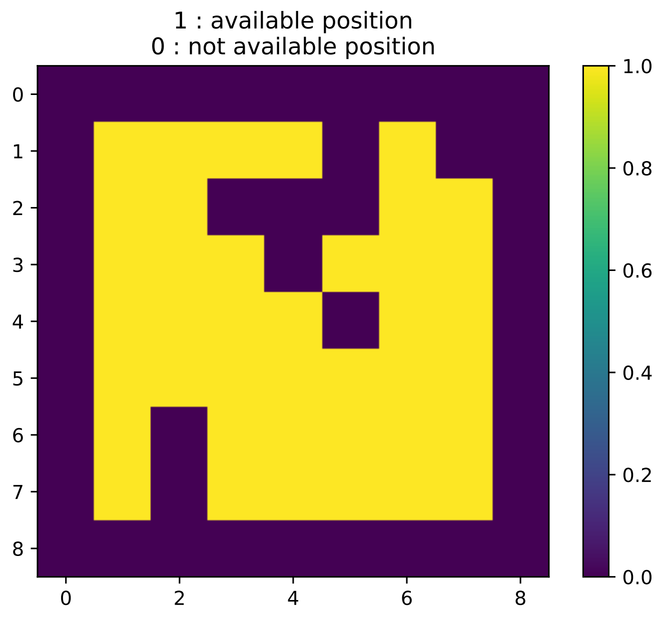


FIGURE 2 – Only the yellow positions are available in this world

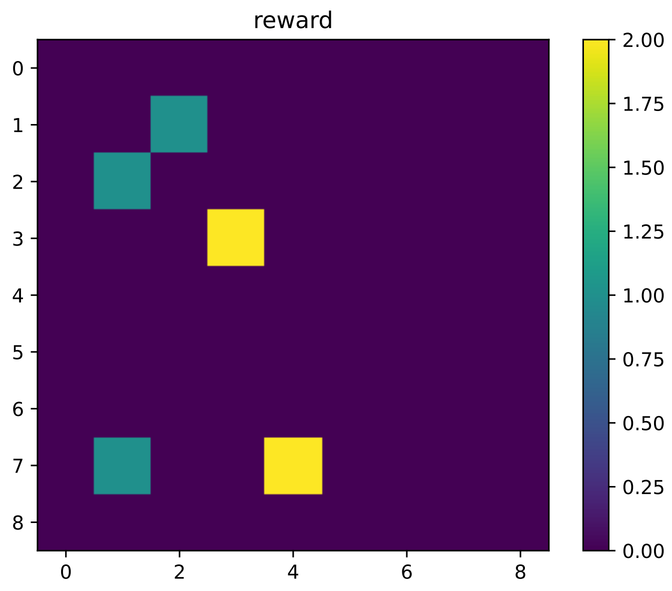


Figure 3 – Available reward in the world (fixed)

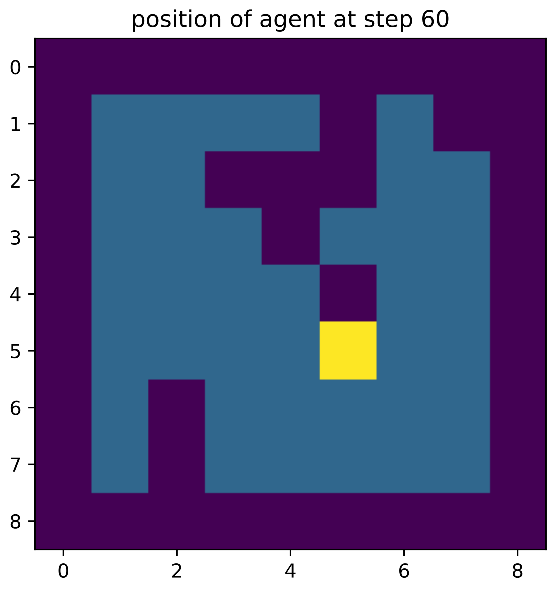


Figure 4 – Agent position

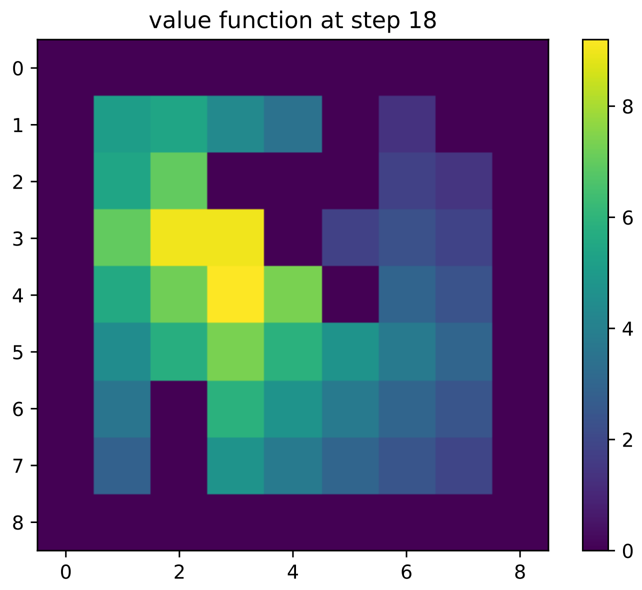


Figure 5 – Estimation of the vlaue function after 18 exploration steps.

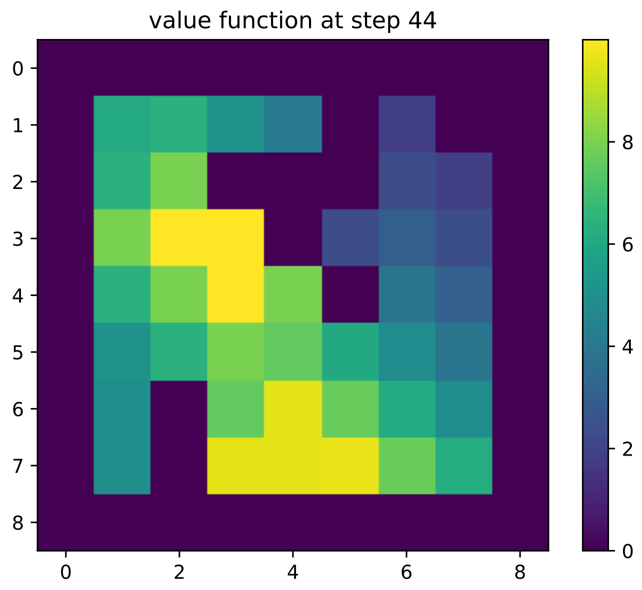
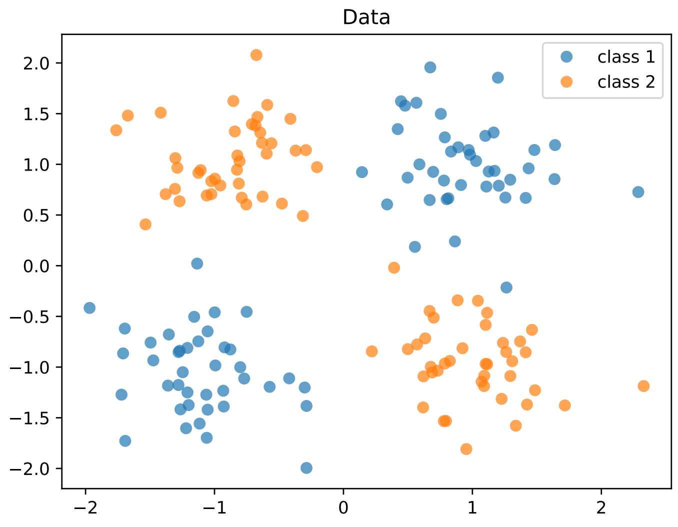


Figure 6 – Estimation of the vlaue function after 44 exploration steps.

3 KERNELS FOR SUPPORT VECTOR MACHINES

We would like to classify these data with a Support vector classifier (SVC).



Obviously, a linear separator will not work to separate these two classes. Hence, we would like to explore kernels and feature maps in order to classify them.

Explore the documentation from scikit-learn and read about the definition of the most commonly used kernels. Try to guess which kernel(s) might work or not in order to classify these data. Only after this stage, test your assumptions by using the library to train a Support vector classifier using different types of kernels and by monitoring the decision function values and a test error.

The data are stored in **exercice_3_svm_kernels/data/**

4 THE HEAVY-BALL METHOD

During the class, we have seen that when optimizing a convex function (such a the empirical risk in a least-squares problem or a logistic regression), the convergence might become very slow when the condition number κ of the Hessian is very large. Some methods exist in order to speed it up, such as **Heavy-ball.** This method consists in adding a **momentum term** to the gradient update term, such as the iteration now writes

$$\theta_{t+1} = \theta_t - \gamma \nabla_\theta f(\theta_t) + \beta(\theta_t - \theta_{t-1}) \tag{1}$$

The update $\theta_{t+1} - \theta_t$ is then a combination of the gradient $\nabla_\theta f(\theta_t)$ and of the previous update $\theta_t - \theta_{t-1}$. The goal of this method it to balance the effet of oscillations in the gradient. The heavy-ball method is called an *inertial method.* When f is a general convex function (not necessary quadratic), some generalizations exist, such as **Nesterov acceleration.**

4.1 Impact on convergence rate for a least squares problem

We note μ the smallest eigenvalue of the Hessian H, and L the largest. Assuming $\mu > 0$, in a least squares problem, it is possible to show that the characteristic convergence time with the heavy-ball momentum term is $\sqrt{\kappa}$ instead of κ. Formally, with the heavy-ball momentum term, we changed the convergence (upper bound) from $\mathcal{O}(\exp(-\frac{2t}{\kappa}))$ to $\mathcal{O}(\exp(-\frac{2t}{\sqrt{\kappa}}))$. If κ is large, which is the case we are interested in, this can be a great improvement.

Remember that $\kappa = \frac{L}{\mu}$. Often, μ will be very small when n or d is large. For instance, in the case of Ridge regression, we have seen in a previous session that for instance, μ can be of order $\mathcal{O}(\frac{1}{\sqrt{n}})$ (see the computation of the optimal regularisation parameter). Hence, κ may be of order $\sqrt{n}$ or higher.

4.2 Simulation

In **heavy_ball/**, use the file **heavy_ball.py** to implement the Heavy-ball method and compare the convergence speed to that of GD. You will need to experiment with γ and β, and might obtain results like figures 7 and 8.

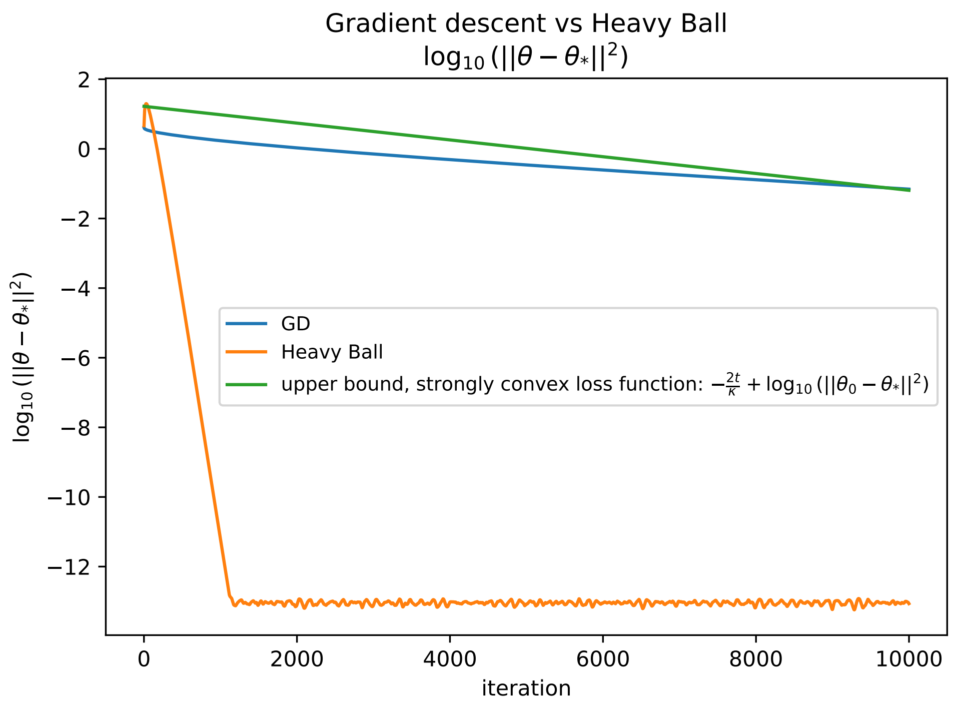


FIGURE 7 – Heavy ball vs GD, semilog scale

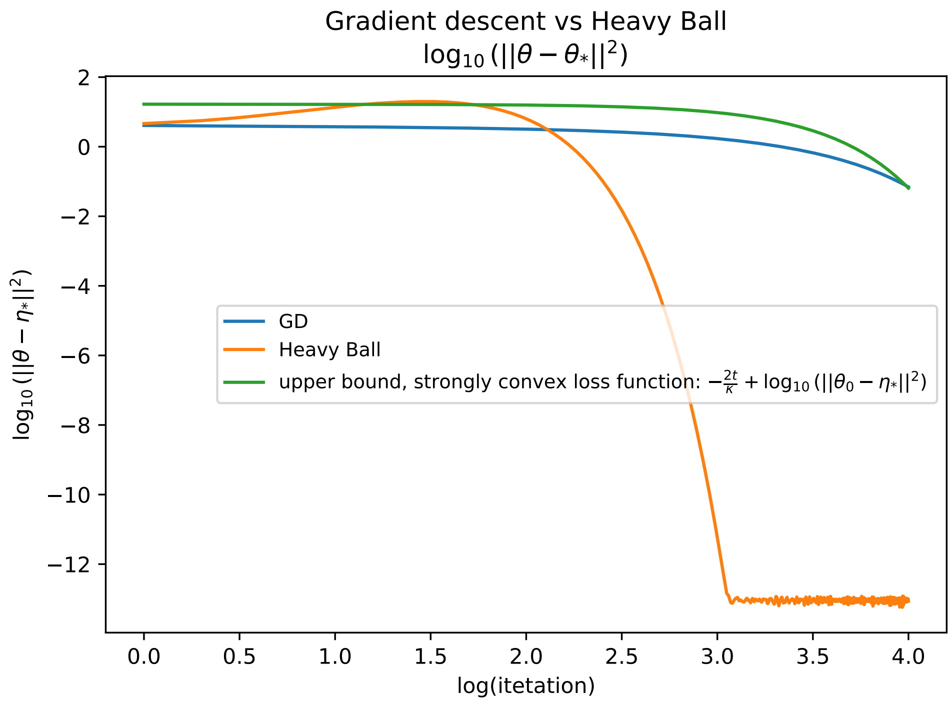
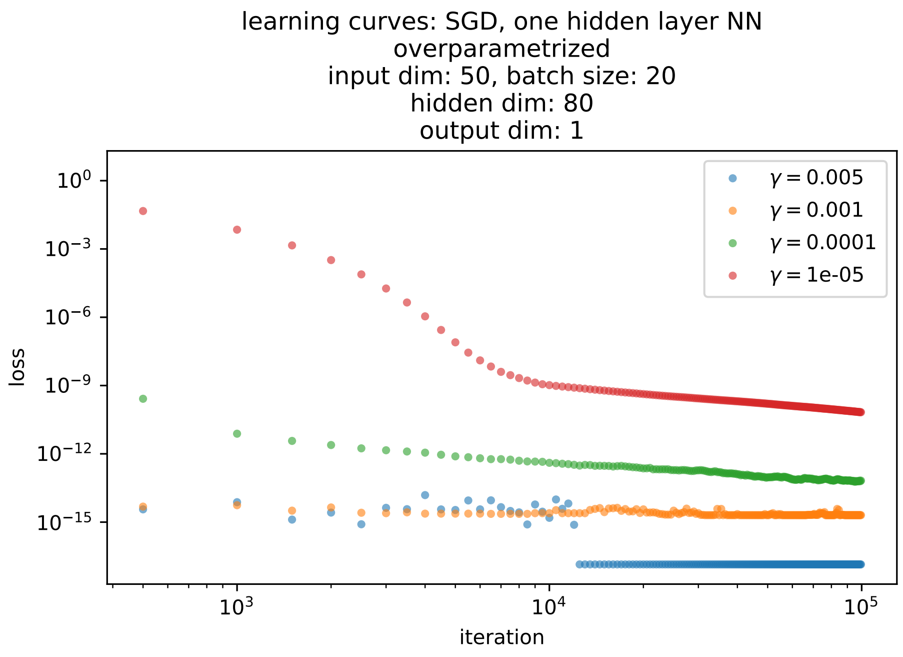
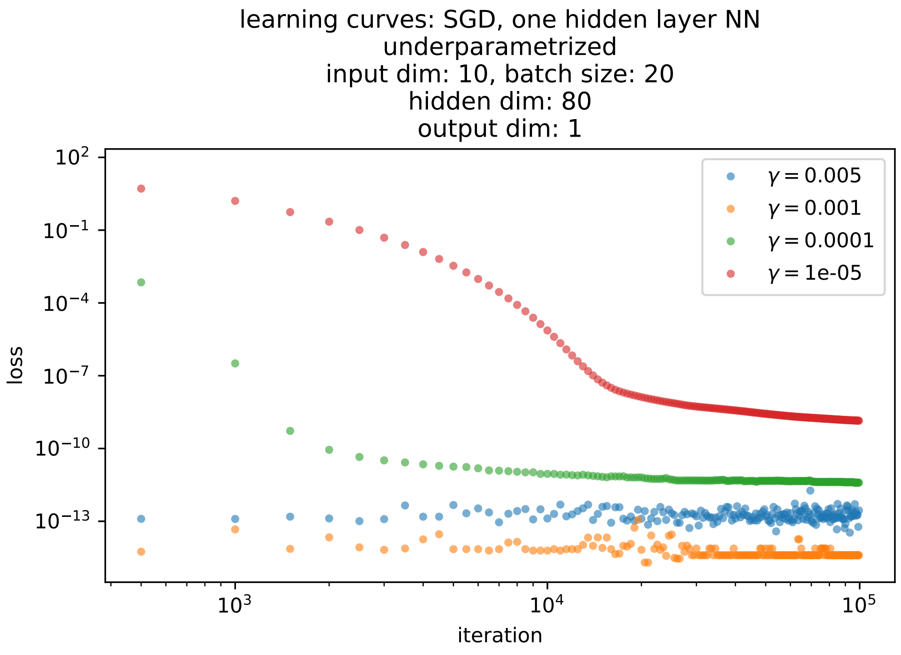


FIGURE 8 – Heavy ball vs GD, logarithmic scale

5 OVERPARAMETRIZED AND UNDERPARAMETRIZED REGIMES




learning curves: SGD, one hidden layer NN
underparametrized
input dim: 10, batch size: 20
hidden dim: 80
output dim: 1
$\gamma = 0.005$
$\gamma = 0.001$
$\gamma = 0.0001$
$\gamma =$ 1e-05
loss
iteration
10^2
10^{-1}
10^{-4}
10^{-7}
10^{-10}
10^{-13}
10^3
10^4
10^5