EL2805 Reinforcement Learning

Computer Lab 1

November 9, 2021

Division of Decision and Control Systems
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology

Deadline: November 28, 2021, 11:59 PM
Instructions (read carefully)

- **(Mandatory)** Solve Problem 1 (Questions (a)-(g)).
- **(Extra)** Solving Problem 1 (Questions (h)-(j)) gives you 1 extra point at the exam. Solving Problem 2 gives you 2 extra point at the exam.
- Work in groups of 2 persons. **Both** students in the group should upload their work to Canvas before the deadline.
- **You must hand-in a zip file containing the following files**:
  1. The python code you used to solve the problems. We expect *at least* 1 file for each problem you solved, named `problem_x.py`, where x is the problem number. Your code should **include both persons' names and personal numbers at the top of the file as a comment**
  2. **In case** you solve Problem 2, a pickle file `weights.pkl`[1] that contains the weights of the Q-function that solves Problem 2.
  3. A joint report where you answer the questions and include relevant figures [2]. **Include both persons' names and personal numbers in the report.**
- **Each plot should have: a legend (if possible), a title, labels for the x and y axes; a caption describing the plot. All the curves in the plots need to be clearly visible.**
- **Name the zip and the report file as follows**:

[1] For more information about saving objects in Python using `Pickle` check `https://wiki.python.org/moin/UsingPickle`

[2] Preferably, use the NeurIPS template for the report:
`https://nips.cc/Conferences/2020/PaperInformation/StyleFiles`

`LASTNAME1-FIRSTNAME1-LASTNAME2-FIRSTNAME2-Lab1.zip`

where `FIRSTNAME1` is the first name of Student 1 in the group (etc.).

- Hand-written solutions will not be corrected. Solutions that have wrong file names or that do not include the code will not be corrected. The `weights.pkl` file will be used to check validity of the solution to Problem 2 in case you have done it.

Problem 1: The Maze and the Random Minotaur

For some unknown reasons you wake up inside a maze (shown in figure 1) in position A. At the same time, there is a minotaur at the exit, in B. The minotaur follows a random walk while staying within the limits of the maze, and can walk inside the walls. This means that for example, if the minotaur is not in a cell at one of the borders of the maze, then it moves to the cell above, below, on the right, and on the left with the same probability 1/4. You cannot walk inside walls, and at a given cell, you may decide to move to an adjacent cell or to stay still. At each step, you observe the position of the minotaur, and decide on a one-step move (up, down, right or left) or not to move. If the minotaur catches you, it will eat you.[3] Your objective is to identify a strategy maximizing the probability of exiting the maze (reaching B) before time T.

Note 1: Neither you nor the minotaur can walk diagonally.
Note 2: The minotaur catches you, if and only if, you are located at the same position, at the same time.

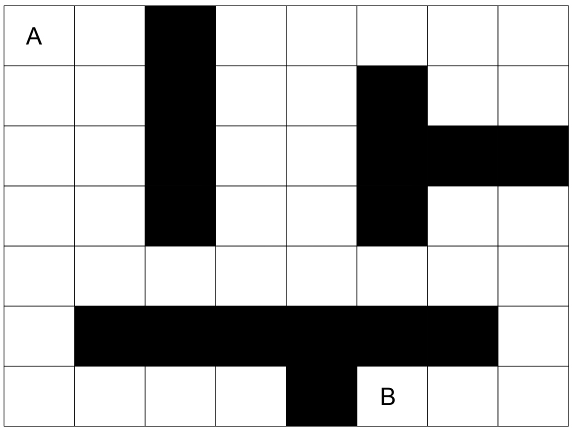

Figure 1: The minotaur's maze.

Basic maze

(a) Formulate the problem as an MDP. Clearly describe the state space, action space, reward and transition probabilities.

Dynamic Programming

(b) Solve the problem: find a policy that maximizes the probability of leaving the maze for $T = 20$. Illustrate this policy.[4]

(c) For $T = 1, \dots, 30$ compute a policy that maximizes the probability of exiting the maze and plot the probability. Is there a difference if the minotaur is allowed to stand still? If so, why?

Value Iteration

(d) You are now poisoned, and need to leave the maze as soon as possible. Due to the poison, your life is geometrically distributed with mean 30. Modify the problem so as to derive a policy maximizing the probability to exit the maze. Motivate your new problem formulation.

(e) Estimate the probability of getting out alive using this policy by simulating 10 000 games.

[3] `https://en.wikipedia.org/wiki/Minotaur`
[4] *Hint:* To illustrate a policy, you could, for example; simulate a game and show the steps taken, plot the action in each player position for a fixed minotaur position, or something else. Be creative.

Q-Learning and Sarsa [Bonus questions (h)-(j)]

(f) Theoretical questions:

1) What does it mean that a learning method is on-policy or off-policy?

2) State the convergence conditions for Q-learning and SARSA.

(g) Consider the scenario in figure 2. Assume you are slightly poisoned, and your life is geometrically distributed with mean 50. Suppose that now the minotaur with probability 35% moves towards you and with probability 65% uniformly at random (remember that the monitaur cannot stay still). To leave the maze you need some keys. The keys are in position C. The agent therefore must learn to first reach C and then get to B. The goal is the same as in (d). Describe how to modify the MDP you formulated in question (d).

BONUS (h) Solve the problem by implementing the Q-learning algorithm using an ε-greedy policy and when exploring, selecting actions uniformly at random.

1) Describe your implementation in pseudo code.

2) Solve the Problem for 2 different values of the exploration parameter ε. Create a plot of the value function over episodes[5] of the initial state, showing the convergence of the algorithm. Use a step size of $1/n(s,a)^{2/3}$, where $n(s,a)$ is the number of times you visited the pair (s,a). Discuss the results, and whether a proper initialization of the Q-values may affect convergence speed. **Note:** Simulate for 50000 episodes.

3) Fix the value of the exploration parameter ε. Show the convergence of the algorithm for 2 different step sizes $1/n(s,a)^{\alpha}$, with $\alpha \in (0.5, 1]$. Discuss the results.

BONUS (i) Solve the problem by implementing the SARSA algorithm.

1) Describe your implementation in pseudo code.

2) Solve for $\varepsilon = 0.2$ and $\varepsilon = 0.1$, using a step size of $1/n(s,a)^{\alpha}$, with $\alpha = 2/3$. Create a plot of the value function over episodes of the initial state. Discuss the results, and whether a proper initialization of the Q-values may affect convergence speed. **Note:** Simulate for 50000 episodes.

3) Now consider the case where the exploration parameter ε decreases in each episode (for example, in episode $k = 1, \ldots$ choose $\varepsilon_k = 1/k^{\delta}$ with $\delta \in (0.5, 1]$). Does convergence improve? Is it better to have $\alpha > \delta$, or the opposite? Argument your answers.

BONUS (j) Estimate the probability of leaving the maze using (1) a policy computed through Q-learning, and (2) another policy computed using SARSA. Are the probabilities close to the Q-value of the initial state (for the respective learning method)? If so/not, explain why.

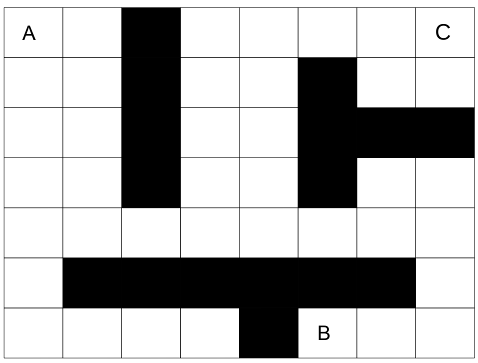

Figure 2: The minotaur's maze with the keys in position C.

[5]An episode terminates when you are either dead, or you leave the maze alive with the keys

Problem 2 (Bonus problem): RL with linear function approximators

1. Background and preliminaries

Reinforcement Learning (RL) in large state/action-spaces most often requires the use of function approximators, such as Neural networks. The idea is to parametrize the state value function of a given policy, the value function or the Q-function using a low dimensional parameter. One possible parametrization consists in expressing these functions as a weighted sum of feature functions. These feature functions are also referred to as *basis functions*, and the resulting method as linear function approximation. A wide variety of basis functions have been used, such as radial basis functions (RBFs), polynomial, and so on. Linear function approximation is attractive because it results in simple update rules (often using gradient descent) and possesses a quadratic error surface with a single minimum (except in degenerate cases). Often, choosing the right function basis is critical. For the problem investigated here, we will use the Fourier basis, a simple linear function approximation scheme that is widely used in applied sciences.

Linear function approximation. In linear function approximation, we approximate a function of interest (the state value function of a policy, the value function or the Q-function) by a linear combination of features of the states using a set of basis functions $\phi_1, \dots, \phi_m$. Functions taking as input the state only (the state value function of a policy, or the value function) are approximated by $V_{\boldsymbol{w}}(s) = \boldsymbol{w}^\top \boldsymbol{\phi}(s)$, where $\boldsymbol{w} = \begin{bmatrix} w_1, \dots, w_m \end{bmatrix}$ and $\boldsymbol{\phi}(s) = \begin{bmatrix} \phi_1(s), \dots, \phi_m(s) \end{bmatrix}$. The learning problem then consists in tuning the vector $\boldsymbol{w}$ so that the approximation is as accurate as possible. Functions taking as input a (state, action) pair (the Q-function) can be approximated by $Q_{\boldsymbol{w}}(s,a) = \boldsymbol{w}_a^\top \boldsymbol{\phi}(s)$ where $\boldsymbol{w} = \begin{bmatrix} \boldsymbol{w}_{a_1}, \dots, \boldsymbol{w}_{a_A} \end{bmatrix}$ (we hence have one vector per action – A is the number of available actions).

Sarsa(λ). For the problem considered in this lab, we will use Sarsa with *eligibility traces* (please refer to Section 12.7 in Sutton's and Barto's book for details). Eligibility traces can be used to unify and generalize Temporal Difference ($\lambda = 0$) and Monte Carlo methods ($\lambda = 1$), where $\lambda \in [0, 1]$ here is the eligibility trace, and should not be confused with the discount factor. Eligibility traces provide a way to compute Monte-Carlo methods in an online fashion. The main benefit is the following: this method allows to correctly update the actions that most contributed to the total reward. This is extremely important in those environments where reward is sparse, like the one we solve here.

The Sarsa(λ) proceeds as follows. For each action a, we maintain an eligibility trace $\boldsymbol{z}_a$, a vector of the same dimension as $\boldsymbol{w}_a$ and initialized at 0. Let $\boldsymbol{z} = \begin{bmatrix} \boldsymbol{z}_{a_1}, \dots, \boldsymbol{z}_{a_A} \end{bmatrix}$.

In step t, let π_t denote the ϵ-greedy policy w.r.t. $Q_{\boldsymbol{w}}$. Generate $a_t, r_t, s_{t+1}, a_{t+1}$ using π_t where under π_t, a_t is the action selected in state s_t, the observed reward is r_t, the next state is s_{t+1}, and a_{t+1} is the action selected in state s_{t+1}. From these observations, we update the eligibility trace $\boldsymbol{z}$ and the parameter $\boldsymbol{w}$ as follows:

$$\boldsymbol{z}_a \leftarrow \begin{cases} \gamma\lambda \boldsymbol{z}_a + \nabla_{\boldsymbol{w}_a} Q_{\boldsymbol{w}}(s_t, a) & \text{if } a = a_t \\ \gamma\lambda \boldsymbol{z}_a & \text{otherwise} \end{cases}, \quad \forall a \in \{a_1, \dots, a_A\}$$

where γ is the discount factor.

A Stochastic Gradient Descent (SGD) is used to update the vector $\boldsymbol{w}$ at time t:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \delta_t \boldsymbol{z},$$

where α is the learning rate and δ_t is the temporal difference error: $\delta_t = r_t + \gamma Q_{\boldsymbol{w}}(s_{t+1}, a_{t+1}) - Q_{\boldsymbol{w}}(s_t, a_t)$. Since we will be training over episodes, it is very important that you remember to reset the eligibility trace at the beginning of each episode! (together with the velocity term $\boldsymbol{v}$ in case you use SGD with momentum, see next section).

Fourier basis. In this problem, we use the Fourier basis. One can read more details in [2] regarding the Fourier basis. We define the p-th order Fourier basis for n variables (the dimensionality of the state s) as follows

$$\phi_i(s) = \cos(\pi \boldsymbol{\eta}_i^\top s), \quad i \in \{1, \dots, m\}$$

where $\boldsymbol{\eta}_i$ is an n-dimensional vector (same as s) $\boldsymbol{\eta}_i = \begin{bmatrix}\eta_{i,1} & \eta_{i,2} & \dots & \eta_{i,n}\end{bmatrix}$ and each $\eta_{i,j}$ takes values in $\{0, 1, \dots, p\}$. Each basis function has a vector $\boldsymbol{\eta}$ that attaches an integer coefficient (less than or equal to p) to each variable in s; the basis set is obtained by systematically varying these coefficients. The vectors $\boldsymbol{\eta}_i$ are **designed by the user**, and capture the interaction between the state variables and the action variables.

As a side note, constraining the vector $\boldsymbol{\eta}_i$ so that only one element is different than 0 enforces variables decoupling. On the other hand, if we want to enforce the coupling between different variables, e.g. x_1 and x_3, then we should enforce $\eta_{i,1}$ and $\eta_{i,3}$ to be different than 0 (i.e., set all the other elements to 0). In domains where the state variables are decoupled it is encouraged to used decoupled basis. For more information, please refer to [1].

2. Tips and tricks

Stabilizing the learning process. While solving the exercise you will most likely experience issues during the training process. As mentioned, it is important that you use Sarsa(λ) for this exercise. Tuning the discount factors γ and the eligibility parameter λ will be left as an exercise, but we would like to suggest some ways to improve the SGD update rule:

- **SGD Modifications.** You may try to implement one of the following two modifications for SGD. These changes bring stability to the learning process and reduce oscillations.

  1. *SGD with Momentum.* This type of gradient update is less susceptible to oscillations in the parameter update. Introduce the velocity term $\boldsymbol{v}$: the SGD step looks like

  $$\begin{aligned}\boldsymbol{v} &\leftarrow m\boldsymbol{v} + \alpha\delta\boldsymbol{e}\\ \boldsymbol{w} &\leftarrow \boldsymbol{w} + \boldsymbol{v}\end{aligned}$$

     where $m \in [0, 1)$ is the momentum parameter, $\boldsymbol{e}$ is the eligibility trace, δ is the temporal difference error, α is the learning rate and $\boldsymbol{w}$ contains the weights of the basis. For $m = 0$ we retrieve the original SGD update. For $m \neq 0$ we get an exponential weighted average of the SGD step, which is less susceptibles to oscillations.

  2. *SGD with Nesterov Acceleration.* It is a slight variation of SGD with momentum, called *Nesterov acceleration*

  $$\begin{aligned}\boldsymbol{v} &\leftarrow m\boldsymbol{v} + \alpha\delta\boldsymbol{e}\\ \boldsymbol{w} &\leftarrow \boldsymbol{w} + m\boldsymbol{v} + \alpha\delta\boldsymbol{e}\end{aligned}$$

     Nesterov acceleration adds a term that is a "correction factor" for the momentum term, and helps reducing oscillations in a "smart" way. The parameters are the same as in SGD with Momentum.

- **Clipping the eligibility trace.** It is common to clip the gradient update to avoid the exploding gradient problem. Clipping means bounding the values of a vector between two pre-defined thresholds. For example, in this problem we suggest you to clip the eligibility trace between -5 and 5 (use `np.clip(z, -5, 5)` where `np` is the NumPy library).

- **Scaling the Fourier basis.** As pointed out in [2], it is better to scale the learning rate for each basis function ϕ_i. Given a basic learning rate α we find that setting the learning rate for ϕ_i to $\alpha_i = \alpha/\|\boldsymbol{\eta}_i\|_2$ performs best (if $\|\boldsymbol{\eta}_i\|_2 = 0$ then $\alpha_i = \alpha$).

- **Reduce the learning rate during training.** It may be useful to reduce the learning rate during training. For example, if the goal is to reach a certain score R, then you may decrease the learning rate of 30% whenever you are "*close*" to R. This ensures that whenever the agent is close to a solution, it does not "run away" from that solution. It is important that you don't decrease the value of the learning rate drastically (otherwise you will get stuck).

3. Task

Your goal is to solve the MountainCar environment[6] using linear function approximators. The MountainCar environment is an example of environment where the state-space is continuous and the action space is discrete. Specifically, the state s is a 2-dimensional variable, where the first dimension s_1 represents position, with $-1.2 \leq s_1 \leq 0.6$ and s_2 represents velocity, with $-0.07 \leq s_2 \leq 0.07$.

There are 3 actions: *push left* (0), *no push* (1) and *push right* (2). The environment is episodic, and you will have to train over multiple episodes. At the beginning of each episode the cart will spawn in a random position between -0.6 and -0.4 at the bottom of the hill, with 0 velocity.

For each action taken you will get a reward of -1 until the goal position (shown in the figure, position 0.5) is reached. You have at most 200 actions to reach the top of the hill. An episode terminates either if: (I) 200 actions have been taken or (II) if the cart reached the goal position. Clearly, the goal is to make the cart reach the flag at the top of the hill.

The system is unknown to you: the dynamics, mass, friction and other parameters of the system are not known. For this reason, you will use a model-free approach, Sarsa(λ), to solve the task.

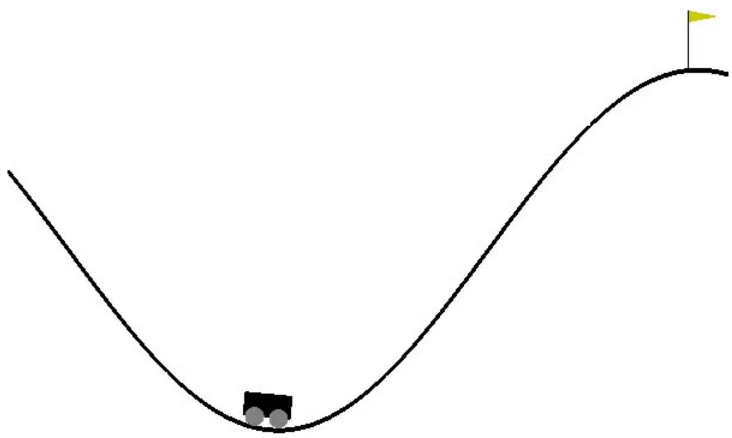

Figure 3: Mountaincar environment. The goal is to reach the flag at the top of the hill.

(a) Check the folder `problem4`. Inside you will find three files:

  (1) `problem4.py`: You will write your code in this file. Get yourself acquainted with the Python code inside the file. Understand the meaning of each variable and how the environment works. *Hint: make sure the state variables are normalized in the* $[0, 1]^2$ *box.*

  (2) `check_solution.py`: You can execute the command `python check_solution.py` to verify validity of the policy you found (check next question).

  (3) `Konidaris2011a.pdf`: reference [1] (you can find more information regarding Fourier basis in this file).

(b) Implement linear function approximation using Fourier basis with $p = 2$ and solve the problem using Sarsa(λ) (it is up to you the design of the various $\boldsymbol{\eta}_i$). Solving the problem means finding a policy π that maximizes the episodic total reward for each initial state s, where the total reward of in state s is given by

$$V^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{T} r(s_t, a_t) | s_0 = s, \pi\right]$$

[6] `https://github.com/openai/gym/wiki/MountainCar-v0`

where T is the episode length and the discount factor γ is assumed to be 1. The problem is solved if your policy is an getting average total reward of at least -135 computed over 50 different episodes. If needed, apply the advices from the previous *Trips and tricks* section. *Hints: you should be able to solve the problem in about $\sim$ 200 episodes, not more than 1000...*

(c) Clearly describe the training process: for how many episodes did you train, which values of the parameter did you use, a short description of the algorithm and the fourier basis and if you used any SGD modification.

(d) Do the following analysis of the training process and the optimal policy:

(1) Plot a figure showing how the episodic total reward changes across episodes during training and analyse it.

(2) Do a 3D plot of the value function of the optimal policy over the state space domain. Try to interpret the plot, does it make sense?

(3) Plot the optimal policy over the state space domain (you should get a 3D plot). Try to interpret the plot, does it make sense?

(4) Did you include $\boldsymbol{\eta} = [0, 0]$ in your basis? Does it make a difference in your opinion?

(5) Compare your agent with an agent taking actions uniformly at random. Plot the episodic total reward over 50 episodes.

(e) Show the average total reward of the policy as a function of α, the learning rate. We suggest to compute the average total reward out of 50 episodes (if possible, show confidence intervals). Repeat the analysis with λ, the eligibility trace. Analyse the plots.

(f) Choose the optimal policy that solves (b). Create a matrix W of dimensions $k \times m$ that contains the weights of the Q-function and a matrix N of dimensions $m \times n$ that contains the various $\boldsymbol{\eta}_i$

$$W = \begin{bmatrix} \boldsymbol{w}_{a_1}^\top \\ \boldsymbol{w}_{a_2}^\top \\ \boldsymbol{w}_{a_3}^\top \end{bmatrix}, \quad N = \begin{bmatrix} \boldsymbol{\eta}_1^\top \\ \vdots \\ \boldsymbol{\eta}_m^\top \end{bmatrix}$$

Create a dictionary object `data={'W':W, 'N':N}` and save it to a file named `weights.pkl` using the library `pickle`[7]. Performance of your agent will be evaluated according to this file. You can execute the command `python check_solution.py` to verify validity of your policy.

References

[1] Konidaris, George. "Value function approximation in reinforcement learning using the Fourier basis." Computer Science Department Faculty Publication Series (2008): 101.

[7] For more information about saving objects in Python check `https://wiki.python.org/moin/UsingPickle`