EL2805 Reinforcement Learning

Computer Lab 1

November 5, 2020

Department of Automatic Control
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology

Deadline: November 29, 2020, 11:59 PM
Instructions (read carefully)

- **(Mandatory)** Solve Problems 1 and 2.
- **(Extra)** You can solve also problem 3 or 4 (or both). Solving problem 3 gives you 1 extra point at the exam (out of 50 points), while solving problem 4 gives you 2 points. Solving both problems 3 and 4 gives you 3 points.
- 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 4, a pickle file `weights.pkl`[1] that contains the weights of the Q-function that solves Problem 4.
  3. A joint report where you answer the questions and include relevant figures [2]. **Include both persons' names and personal numbers in the report.**
- **Name the zip and the report file as follows**:

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

[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`

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 4 in case you have done it.

Problem 1:
The Maze and the Random Minotaur

Consider the maze in Figure 1. You enter the maze in A and at the same time, the minotaur enters in B. The minotaur follows a random walk while staying within the limits of the maze. The minotaur's walk goes through walls (which obviously you cannot do). 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, he 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.

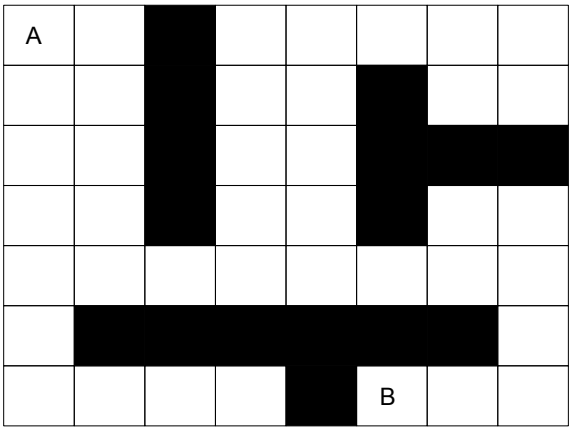

Figure 1: The minotaur's maze.

(a) Formulate the problem as an MDP.

(b) Solve the problem, and illustrate an optimal policy for $T = 20$.[4] Plot the maximal probability of exiting the maze as a function of T. Is there a difference if the minotaur is allowed to stand still? If so, why?

(c) Assume now that your life is geometrically distributed with mean 30. Modify the problem so as to derive a policy minimizing the expected time to exit the maze. Motivate your new problem formulation. 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.

Problem 2:
Robbing Banks

At time 0, you are robbing Bank 1 (see Figure 4), and the police gets alerted and starts chasing you from the point PS (Police Station). You observe where the police is, and decide in each step either to move up, left, right, down or to stay where you are. Each time you are at a bank, and the police is not there, you collect a reward of 10 SEK. If the police catches you, you loose 50 SEK, and the game is reinitialised (you go back to Bank 1, and the police goes back to the PS).

The police always chases you, but moves randomly in your direction. More precisely, assume that the police and you are on the same line, and without loss of generality, that the police is on your right; then the police moves up, down and left with probability 1/3. Similarly, when the police and you are on the same column, and when your are above the police, then the police moves up, right and left with probability 1/3. When the police and you are not on the same line, nor on the same column, say the police is below you and on your right, then the police moves up and left with probability 1/2.

The rewards are discounted at rate $\lambda \in (0, 1)$, and your objective is to maximise your average discounted reward.

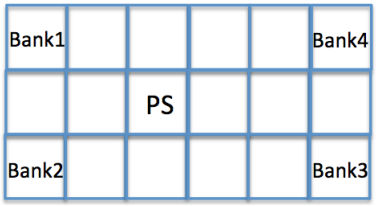

Figure 2: The city where you are robbing banks.

(a) Formulate the problem as an MDP.

(b) Solve the problem, and display the value function (evaluated at the initial state) as a function of λ. Illustrate an optimal policy for different values of λ – comment on the behaviour.

Problem 3:
Bank Robbing (Reloaded)

You are a bank robber trying to heist the bank of an unknown town. You enter the town at position A (see Figure 3), the police starts from the opposite corner, and the bank is at position B. For each round spent in the bank, you receive a reward of 1 SEK. The police walks randomly (that is, uniformly at random up, down, left or right) across the grid and whenever you are caught (you are in the same cell as the police) you lose 10 SEK.

You are new in town, and hence oblivious to the value of the rewards, the position of the bank, the starting point and the movement strategy of the police. Before you take an action (move up, down, left, right or stand still), you can observe both your position and that of the police. Your task is to develop an algorithm learning the policy that maximizes your total discounted reward for a discount factor $\lambda = 0.8$.

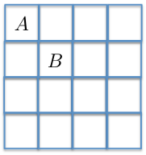

Figure 3: The unknown town.

(a) Solve the problem by implementing the Q-learning algorithm exploring actions uniformly at random. Create a plot of the value function over time (in particular, for the initial state), showing the convergence of the algorithm. **Note:** Expect the value function to converge after roughly 10 000 000 iterations (for step size $1/n(s,a)^{2/3}$, where $n(s,a)$ is the number of updates of $Q(s,a)$).

(b) Solve the problem by implementing the SARSA algorithm using ε-greedy exploration (initially $\varepsilon = 0.1$). Show the convergence for different values of ε.

Problem 4:
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 $\boldsymbol{w} = \big[w_1, \dots, w_m\big]$ and $\boldsymbol{\phi}(s) = \big[\phi_1(s), \dots, \phi_m(s)\big]$. 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 $\boldsymbol{w} = \big[\boldsymbol{w}_{a_1}, \dots, \boldsymbol{w}_{a_A}\big]$ (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 $\boldsymbol{z} = \big[\boldsymbol{z}_{a_1}, \dots, \boldsymbol{z}_{a_A}\big]$.

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

     $$\boldsymbol{v} \leftarrow m\boldsymbol{v} + \alpha\delta\boldsymbol{e}$$
     $$\boldsymbol{w} \leftarrow \boldsymbol{w} + \boldsymbol{v}$$

     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*

     $$\boldsymbol{v} \leftarrow m\boldsymbol{v} + \alpha\delta\boldsymbol{e}$$
     $$\boldsymbol{w} \leftarrow \boldsymbol{w} + m\boldsymbol{v} + \alpha\delta\boldsymbol{e}$$

     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[5] 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.

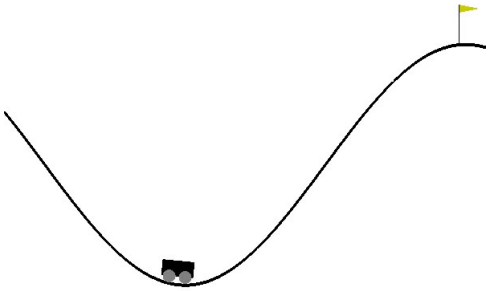

Figure 4: Mountaincar environment. The goal is to reach the flag at the top of the hill.

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

  (a) `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.*

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

  (c) `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

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

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 over 30 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...*

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

(c) 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) Plot a figure showing how the episodic total reward changes across episodes during training. Plot the value function of the optimal policy.

(d) What happens if you lower/increase α (learning rate), λ (eligibility trace parameter)? For each parameter, try to increase/lower it and keep the other one fixed. Show how the training process looks like for each scenario (you should have 4 plots).

(e) Choose the 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`[6]. Performance of your agent will be evaluated according to this file. You can execute the command `python check_solution.pyc` 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.

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