EL2805 Reinforcement Learning

Computer Lab 0

October 18, 2020

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

Environment setup: Make sure you have read the `el2805_lab_instructions.pdf` file.

Key concepts: Markov Decision Processes, Dynamic Programming, Value Iteration, PyTorch programming.

Instructions: The following problems are proposed to get familiar with python environment, and to apply some of the basic concepts of the course. A solution will be published on Canvas.

Problem 1:
Shortest Path in the Maze

In this problem, consider the maze in Figure 1. The black cells represent obstacles or walls. The player enters the maze at A. By convention, the cells are labeled by their position on the x, y axis, so that $A = (0, 0)$ and $B = (5, 5)$. At each time step, the player can make a one-step move (up, down, right or left) or stay in her position. Her only objective, is to find the shortest path to the exit B, given full knowledge of the maze.

Note 1: The player cannot walk diagonally.

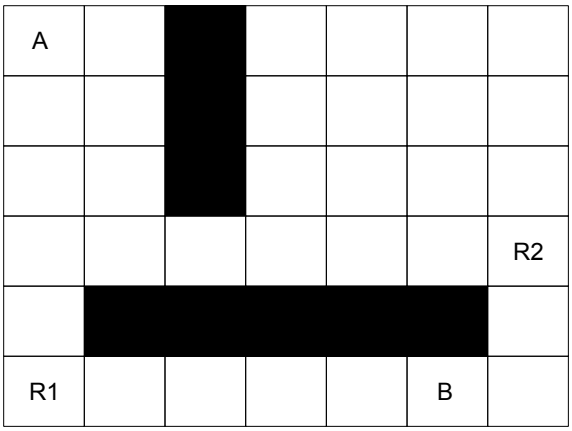

Figure 1: The maze.

(a) Formulate the problem as an MDP.

- Specify the state space $\mathcal{S}$.
- Specify the action space $\mathcal{A}$.
- Specify the transition probabilities $\mathcal{P}$.
- Specify the rewards $\mathcal{R}$.

(b) Solve the formulated MDP using dynamic programming, motivating your choice of the time horizon T and illustrate the shortest path solution, i.e., show the optimal action taken at each cell.

(c) Solve the formulated MDP using value iteration, and show the evolution of the values of each state.

(d) Solve the following modified problem. Every time the player visits the cell $R1$, she must stay there for 6 consecutive rounds with probability 0.5 (she can move as usual with probability 0.5). Similarly, when she visits $R2$, she must stay for 1 round with probability 0.5.

Hint: Modify the rewards only, so that they become random variables.

Problem 2:
Plucking berries

We consider a variation of Problem 1. Given the same maze as in Figure 1, the player now wants to find the most *rewarding* plucking path within T steps, starting at A, and given the following weight matrix W:

$$W = \begin{bmatrix} 0 & 1 & -\infty & 10 & 10 & 10 & 10 \\ 0 & 1 & -\infty & 10 & 0 & 0 & 10 \\ 0 & 1 & -\infty & 10 & 0 & 0 & 10 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & -\infty & -\infty & -\infty & -\infty & -\infty & 10 \\ 0 & 0 & 0 & 0 & 0 & 11 & 10 \end{bmatrix}.$$

The weight of each element is associated with the reward obtained upon moving on that cell.

(a) Modify your MDP formulation of Problem 1 so that it matches this setting.

(b) Solve the new MDP using dynamic programming, and illustrate the maximum value path. Do this for every T from 12 to 20.

(c) Solve the formulated MDP using value iteration, and show the obtained maximum value path.

Problem 3:
Pytorch for RL

In this exercise we will not solve any problem, but we will start to get used to PyTorch, Gym and other concepts used in Deep Reinforcement Learning (DRL).

(a) Check the file `lab0_p3.py` and run it. Get acquainted with the code. You should see a cartpole being controlled by a random agent.

(b) In DRL a commonly used tool is the *experience replay buffer*. It is a list of tuples, where each tuple x is 5-dimensional $x = (s, a, r, s', d)$, where s is a state, a is an action, r is the reward that you got after taking action a in state s, s' is the state that occured after taking action a in state s and d is a boolean variable that indicates if the problem terminated. You will implement this experience replay buffer:

- Create a buffer: the maximum size should be fixed, and whenever the buffer is full you should discard older elements in favour of new ones. We suggest using the class `deque` from the library `collections` (just do `from collections import deque`).
- Change the code so that after every action taken in the environment you append the tuple (s, a, r, s', d) to the buffer.
- Create a function that randomly samples N elements from this buffer.

(c) We will now implement a neural network. Create a Neural Network class that satisfies the following:

- The input size should be equal to the state dimensionality of the MDP.
- The first layer should be fully connected with 8 neurons. The activation of the first layer should be ReLU.
- Create the output layer: it should have as many neurons as the number of actions. Do not add any activation layer.
- Create an appropriate `forward` function inside the neural network class.

(d) We will now take actions according to the neural network we just defined.

- Comment out the line where we take actions uniformly at random.
- Encapsulate the state inside a list and create a tensor with disabled gradient computation, i.e., `state_tensor=torch.tensor([state], requires_grad=False)`, and pass `state_tensor` to the neural network. The output of the neural network should be a tensor x of dimensions $[1, m]$ where m is the number of actions.
- Choose the action according to $\text{argmax}_a\, x_a$. In PyTorch you can do it by typing the following command `action = x.max(1)[1]`. `max(1)` means that you are checking the maximum along the first axis (second dimension, since the first dimension is the 0-axis), and the notation `[1]` means that you are interested in the second output of this function, which is the index of the maximum element. End by writing `action = action.item()`, which returns the value of the action tensor.

(e) Finally, we end the exercise by performing backpropagation.

- Create an optimizer that will act on the neural network parameters. You can use `Adam` or `SGD` from the `torch.optim` library. For example, you can write `optim = torch.optim.Adam(neural_network.parameters(), lr=0.01)`, where `lr` is the learning rate and `neural_network` is the network that you created in (d).
- Now we will train the network so that it minimizes some quadratic loss function. After taking an action, set the gradients to zero by using `optim.zero_grad()` and sample 3

random experiences $\{(s_i, a_i, r_i, s'_i, d_i)\}_{i=1}^{3}$ from the buffer that you created (observe that if you have less than 3 experiences in your buffer then you skip the training step). Try to minimize the loss $\sum_i x_{a_i}^2$: feed each s_i to the network, get an output x and choose a value from x according to a_i. You should have one value for each experience i, therefore in the end you should have a vector z of 3 values. Create a zero vector y with 3 elements using Pytorch. Use the command `loss=torch.nn.functional.mse_loss(z,y)` function to minimize the loss defined before. End by performing `loss.backward()` to compute the gradient and `optim.step()` to perform backpropagation. You can also clip the gradient to avoid the *exploding gradient* phenomenon by typing
`torch.nn.utils.clip_grad_norm_(neural_network.parameters(), 1)`, where 1 is a commonly used value.