Assignment 5

May 3, 2023

Binsha Nazar Othupalliparambu Nazar 10 h

Julia Szczepaniak 3 h

DAT405/DIT407 Introduction to Data Science and AI

2022-2023, Reading Period 4

Assignment 5: Reinforcement learning and classification

Hints: You can execute certain linux shell commands by prefixing the command with `!`. You can insert Markdown cells and code cells. The first you can use for documenting and explaining your results the second you can use writing code snippets that execute the tasks required.

This assignment is about **sequential decision making** under uncertainty (Reinforcement learning). In a sequential decision process, the process jumps between different states (the environment), and in each state the decision maker, or agent, chooses among a set of actions. Given the state and the chosen action, the process jumps to a new state. At each jump the decision maker receives a reward, and the objective is to find a sequence of decisions (or an optimal policy) that maximizes the accumulated rewards.

We will use **Markov decision processes** (MDPs) to model the environment, and below is a primer on the relevant background theory.

- To make things concrete, we will first focus on decision making under **no** uncertainity (question 1 and 2), i.e, given we have a world model, we can calculate the exact and optimal actions to take in it. We will first introduce **Markov Decision Process (MDP)** as the world model. Then we give one algorithm (out of many) to solve it.
- (Optional) Next we will work through one type of reinforcement learning algorithm called Q-learning (question 3). Q-learning is an algorithm for making decisions under uncertainity, where uncertainity is over the possible world model (here MDP). It will find the optimal policy for the **unknown** MDP, assuming we do infinite exploration.
- Finally, in question 4 you will be asked to explain differences between reinforcement learning and supervised learning and in question 5 write about decision trees and random forests.

Primer

Decision Making

The problem of **decision making under uncertainty** (commonly known as **reinforcement learning**) can be broken down into two parts. First, how do we learn about the world? This involves both the problem of modeling our initial uncertainty about the world, and that of drawing conclusions from evidence and our initial belief. Secondly, given what we currently know about the world, how should we decide what to do, taking into account future events and observations that may change our conclusions? Typically, this will involve creating long-term plans covering possible future eventualities. That is, when planning under uncertainty, we also need to take into account what possible future knowledge could be generated when implementing our plans. Intuitively, executing plans which involve trying out new things should give more information, but it is hard to tell whether this information will be beneficial. The choice between doing something which is already known to produce good results and experiment with something new is known as the **exploration-exploitation dilemma**.

The exploration-exploitation trade-off

Consider the problem of selecting a restaurant to go to during a vacation. Lets say the best restaurant you have found so far was **Les Epinards**. The food there is usually to your taste and satisfactory. However, a well-known recommendations website suggests that **King's Arm** is really good! It is tempting to try it out. But there is a risk involved. It may turn out to be much worse than **Les Epinards**, in which case you will regret going there. On the other hand, it could also be much better. What should you do? It all depends on how much information you have about either restaurant, and how many more days you'll stay in town. If this is your last day, then it's probably a better idea to go to **Les Epinards**, unless you are expecting **King's Arm** to be significantly better. However, if you are going to stay there longer, trying out **King's Arm** is a good bet. If you are lucky, you will be getting much better food for the remaining time, while otherwise you will have missed only one good meal out of many, making the potential risk quite small.

Markov Decision Processes

Markov Decision Processes (MDPs) provide a mathematical framework for modeling sequential decision making under uncertainty. An *agent* moves between *states* in a *state space* choosing *actions* that affects the transition probabilities between states, and the subsequent *rewards* recieved after a jump. This is then repeated a finite or infinite number of epochs. The objective, or the *solution* of the MDP, is to optimize the accumulated rewards of the process.

Thus, an MDP consists of five parts:

- Decision epochs: $t = 1, 2, ..., T$, where $T \leq \infty$
- State space: $S = \{s_1, s_2, ..., s_N\}$ of the underlying environment
- Action space $A = \{a_1, a_2, ..., a_K\}$ available to the decision maker at each decision epoch
- Transition probabilities $p(s_{t+1}|s_t, a_t)$ for jumping from state s_t to state s_{t+1} after taking action a_t
- Reward functions $R_t = r(a_t, s_t, s_{t+1})$ resulting from the chosen action and subsequent transition

A *decision policy* is a function $\pi : s \to a$, that gives instructions on what action to choose in each state. A policy can either be *deterministic*, meaning that the action is given for each state, or *randomized* meaning that there is a probability distribution over the set of possible actions for each state. Given a specific policy π we can then compute the the *expected total reward* when starting in a given state $s_1 \in S$, which is also known as the *value* for that state,

$$V^{\pi}(s_1) = E\left[\sum_{t=1}^{T} r(s_t, a_t, s_{t+1})|s_1\right] = \sum_{t=1}^{T} r(s_t, a_t, s_{t+1})p(s_{t+1}|a_t, s_t)$$

where $a_t = \pi(s_t)$. To ensure convergence and to control how much credit to give to future rewards, it is common to introduce a *discount factor* $\gamma \in [0, 1]$. For instance, if we think all future rewards should count equally, we would use $\gamma = 1$, while if we value near-future rewards higher than more distant rewards, we would use $\gamma < 1$. The expected total *discounted* reward then becomes

$$V^{\pi}(s_1) = \sum_{t=1}^{T} \gamma^{t-1} r(s_t, a_t, s_{t+1})p(s_{t+1}|s_t, a_t)$$

Now, to find the *optimal* policy we want to find the policy π^* that gives the highest total reward $V^*(s)$ for all $s \in S$. That is, we want to find the policy where

$$V^*(s) \geq V^{\pi}(s), s \in S$$

To solve this we use a dynamic programming equation called the *Bellman equation*, given by

$$V(s) = \max_{a \in A}\left\{\sum_{s' \in S} p(s'|s, a)(r(s, a, s') + \gamma V(s'))\right\}$$

It can be shown that if π is a policy such that V^{π} fulfills the Bellman equation, then π is an optimal policy.

A real world example would be an inventory control system. The states could be the amount of items we have in stock, and the actions would be the amount of items to order at the end of each month. The discrete time would be each month and the reward would be the profit.

Question 1

The first question covers a deterministic MPD, where the action is directly given by the state, described as follows:

- The agent starts in state **S** (see table below)
- The actions possible are **N** (north), **S** (south), **E** (east), and **W** west.
- The transition probabilities in each box are deterministic (for example P(s'|s,N)=1 if s' north of s). Note, however, that you cannot move outside the grid, thus all actions are not available in every box.
- When reaching **F**, the game ends (absorbing state).
- The numbers in the boxes represent the rewards you receive when moving into that box.
- Assume no discount in this model: $\gamma = 1$

-1	1	**F**
0	-1	1
-1	0	-1
S	-1	1

Let (x, y) denote the position in the grid, such that $S = (0, 0)$ and $F = (2, 3)$.

1a) What is the optimal path of the MDP above? Is it unique? Submit the path as a single string of directions. E.g. NESW will make a circle.

Answer 1a: The optimal path that we found is EENNN but it is not the unique one as we say. We can also take the path EENNWNE which takes another 2 more moves but reward is same for both (which is 0, given that gamma is equal to 0).

1b) What is the optimal policy (i.e. the optimal action in each state)? It is helpful if you draw the arrows/letters in the grid.

Answer 1b:An optimal policy lists the best actions to take at each state S for the best reward over time. There can be more than one optimal policy in a MDP : all optimal policy achieve the same optimal value function. Here is the optimal policy of the MDP above :

(0,0) : N/E is -1 (0,1) : E is 1 (0,2) : N/W is -1 (1,0) : N/E is 0 (1,1) : N/W/S/E is -1 (1,2) : N/S is 1 (2,0) : N/S/E is -1 (2,1) : N/E is 1 (2,2) : N is 0 (3,0) : E is 1 (3,1) : E is 0 (3,2) : absorbing state (no possible actions)

1c) What is expected total reward for the policy in 1a)?

Answer 1c: The expected total reward fot the policy defined in 1b is 0. We followed several alternative paths and the maximum reward found to be -1 + 1 - 1 + 1 = 0.

Value Iteration

For larger problems we need to utilize algorithms to determine the optimal policy π^*. *Value iteration* is one such algorithm that iteratively computes the value for each state. Recall that for a policy to be optimal, it must satisfy the Bellman equation above, meaning that plugging in a given candidate V^* in the right-hand side (RHS) of the Bellman equation should result in the same V^* on the left-hand side (LHS). This property will form the basis of our algorithm. Essentially, it can be shown that repeated application of the RHS to any intial value function $V^0(s)$ will eventually lead to the value V which statifies the Bellman equation. Hence repeated application of the Bellman equation will also lead to the optimal value function. We can then extract the optimal policy by simply noting what actions that satisfy the equation.

The process of repeated application of the Bellman equation is what we here call the *value iteration* algorithm. It practically procedes as follows:

```
epsilon is a small value, threshold
for x from i to infinity
do
    for each state s
    do
        V_k[s] = max_a Σ_s' p(s|s,a)*(r(a,s,s) + γ*V_k1[s])
        end
        if  |V_k[s]-V_k-1[s]| < epsilon for all s
            for each state s,
            do
                π(s)=argmax_a _s p(s|s,a)*(r(a,s,s) + γ*V_k1[s])
                return π, V_k
            end
    end
```

Example: We will illustrate the value iteration algorithm by going through two iterations. Below is a 3x3 grid with the rewards given in each state. Assume now that given a certain state s and action a, there is a probability 0.8 that that action will be performed and a probability 0.2 that no action is taken. For instance, if we take action **E** in state (x, y) we will go to $(x+1, y)$ 80 percent of the time (given that that action is available in that state), and remain still 20 percent of the time. We will use have a discount factor $\gamma = 0.9$. Let the initial value be $V^0(s) = 0$ for all states $s \in S$.

Reward:

0	0	0
0	10	0
0	0	0

Iteration 1: The first iteration is trivial, $V^1(s)$ becomes the $\max_a \sum_{s'} p(s'|s, a)r(s, a, s')$ since V^0 was zero for all s'. The updated values for each state become

0	8	0
8	2	8
0	8	0

Iteration 2:

Staring with cell (0,0) (lower left corner): We find the expected value of each move:
Action **S**: 0
Action **E**: 0.8(0 + 0.9 * 8) + 0.2(0 + 0.9 * 0) = 5.76
Action **N**: 0.8(0 + 0.9 * 8) + 0.2(0 + 0.9 * 0) = 5.76
Action **W**: 0

Hence any action between **E** and **N** would be best at this stage.

Similarly for cell (1,0):

Action **N**: 0.8(10 + 0.9 * 2) + 0.2(0 + 0.9 * 8) = 10.88 (Action **N** is the maximizing action)

Similar calculations for remaining cells give us:

5.76	10.88	5.76
10.88	8.12	10.88
5.76	10.88	5.76

Question 2

2a) Code the value iteration algorithm just described here, and show the converging optimal value function and the optimal policy for the above 3x3 grid. Make sure to consider that there may be several equally good actions for a state when presenting the optimal policy.

[1]:
```
import numpy as np

# Parameters
iterations = 0 # Number of iterations the algorithm should make
gamma = 0.9 # Reward coefficient
actionTaken = 0.8 # Probability that an action is taken
epsilon = 0.0001 # Convergence threshold -maximum difference between the current␣
 ↪state values and the new state values
#smaller convergence threshold leads to more accurate results


# Original matrices
rewards = np.array([[0,0,0],[0,10,0],[0,0,0]]) # Rewards
states = np.array([[0.0,0.0,0.0],[0.0,0.0,0.0],[0.0,0.0,0.0]]) # Environment
new_states = np.array([[0.0,0.0,0.0],[0.0,0.0,0.0],[0.0,0.0,0.0]]) # New␣
 ↪environment (for computing purpose)

# Computing function
def computeValue(currentReward, nextReward, currentStateValue, nextStateValue,␣
 ↪gammaValue, actionTaken):
    v =␣
 ↪actionTaken*(nextReward+gammaValue*nextStateValue)+(1-actionTaken)*(currentReward+gammaValue*
    return v

# Updating iterations
while True:

    for i in range(0,states.shape[0]):

        for j in range(states.shape[1]):

            # Values reset
            north, east, west, south = 0.0,0.0,0.0,0.0

            # N border check
            if i != 0:
                # Computes north value
                north = computeValue(rewards[i,j], rewards[i-1,j], states[i,j],␣
 ↪states[i-1,j], gamma, actionTaken)

            #E border check
            if j != (rewards.shape[0] - 1):
                # Computes east value
                east = computeValue(rewards[i,j], rewards[i,j+1], states[i,j],␣
↪states[i,j+1], gamma, actionTaken)

            #W border check
            if j != 0:
                # Computes west value
                west = computeValue(rewards[i,j], rewards[i,j-1], states[i,j],␣
↪states[i,j-1], gamma, actionTaken)

            #S border check
            if i != (rewards.shape[1] - 1):
                # Computes south value
                south = computeValue(rewards[i,j], rewards[i+1,j], states[i,j],␣
↪states[i+1,j], gamma, actionTaken)

            # Returns max value in the new states array
            new_states[i,j] = max(north, east, south, west)

    #Calculates absolute difference between computed V and previous V values
    threshold_state = np.absolute(states - new_states)

    # Threshold condition to check
    is_under_threshold = np.all((threshold_state < epsilon))

    # Copy new states array into states array to perform other states values␣
↪update (or to print final output)
    np.copyto(states, new_states)
    iterations+=1 # Adds 1 to iteration counter

    # Threshold condition
    if is_under_threshold:
        print('Converging optimal value = '+str(iterations)+'\n')
        break

print('Optimal policy values =\n')
print(states) # Shows final states matrix
```

```
Converging optimal value = 104

Optimal policy values =

[[45.61205232 51.9471806  45.61205232]
 [51.9471806  48.05107671 51.9471806 ]
 [45.61205232 51.9471806  45.61205232]]
```

The optimal policy can be defined based on the optimal value function. Each element in the matrix indicates which neighboring cell to move to in order to maximize the expected total reward. The optimal policy matrix is (each case indicates what case "i,j" to go next) :

[[(0,1) (1,1) (0,1)]

[(1,1) (0,1) (1,1)]

[(1,0) (1,1) (1,2)]]

if the agent is at position (0,0), the optimal policy is to move to position (0,1) because this results in the highest expected total reward. Similarly, if the agent is at position (1,1), the optimal policy is to stay in the same position, as this also results in the highest expected total reward.

2b) Explain why the result of 2a) does not depend on the initial value V_0.

Answer 2b: At each iteration, the values of the rewards progressively decrease (gamma factor). Regardless of the initial values chosen for V, the optimal policy found by our algorithm will always converge to the same one. If the initial values are large, the algorithm will have to perform more iterations to achieve convergence (convergence is finite when the difference between the values of the matrices Vt and Vt-1 are less than epsilon).

2c) Describe your interpretation of the discount factor γ. What would happen in the two extreme cases $\gamma = 0$ and $\gamma = 1$? Given some MDP, what would be important things to consider when deciding on which value of γ to use?

Answer 2c: The discount factor is a value that determines the importance of future rewards compared to immediate rewards. When the discount factor is set to 0, the agent only cares about the immediate rewards and doesn't take future rewards into account. This means that the agent will only try to consider the rewards obtained in the current state and ignore any rewards that can be obtained in future states. As a result, the agent won't try to maximize its long-term performance. While just in case when the discount factor is set to 1, the agent cares equally about immediate and future rewards which means that the agent will take into account all the rewards that can be obtained in future states when deciding on its actions. As a result, the agent will behave with a long-term focus and will try to maximize its long-term performance and also will be ready to sacrifice immediate rewards. γ value may depends on either nature of the rewards or how far into future the agent needs to plan. Small value of γ is suitable for immediate rewards while larger value is ideal for future rewards. Also when the rewards are stable for a period of time, larger γ value is good as agent can have long term plans.

Reinforcement Learning (RL) (Theory for optional question 3)

Until now, we understood that knowing the MDP, specifically $p(s'|a,s)$ and $r(s,a,s')$ allows us to efficiently find the optimal policy using the value iteration algorithm. Reinforcement learning (RL) or decision making under uncertainity, however, arises from the question of making optimal decisions without knowing the true world model (the MDP in this case).

So far we have defined the value function for a policy through V^π. Let's now define the *action-value function*

$$Q^\pi(s,a) = \sum_{s'} p(s'|a,s)[r(s,a,s') + \gamma V^\pi(s')]$$

The value function and the action-value function are directly related through

$$V^\pi(s) = \max_a Q^\pi(s,a)$$

i.e, the value of taking action a in state s and then following the policy π onwards. Similarly to the value function, the optimal Q-value equation is:

$$Q^*(s,a) = \sum_{s'} p(s'|a,s)[r(s,a,s') + \gamma V^*(s')]$$

and the relationship between $Q^*(s,a)$ and $V^*(s)$ is simply

$$V^*(s) = \max_{a \in A} Q^*(s,a).$$

Q-learning Q-learning is a RL-method where the agent learns about its unknown environment (i.e. the MDP is unknown) through exploration. In each time step t the agent chooses an action a based on the current state s, observes the reward r and the next state s', and repeats the process in the new state. Q-learning is then a method that allows the agent to act optimally. Here we will focus on the simplest form of Q-learning algorithms, which can be applied when all states are known to the agent, and the state and action spaces are reasonably small. This simple algorithm uses a table of Q-values for each (s,a) pair, which is then updated in each time step using the update rule in step $k+1$

$$Q_{k+1}(s,a) = Q_k(s,a) + \alpha\left(r(s,a) + \gamma \max\{Q_k(s',a')\} - Q_k(s,a)\right)$$

where γ is the discount factor as before, and α is a pre-set learning rate. It can be shown that this algorithm converges to the optimal policy of the underlying MDP for certain values of α as long as there is sufficient exploration. For our case, we set a constant $\alpha = 0.1$.

OpenAI Gym We shall use already available simulators for different environments (worlds) using the popular OpenAI Gym library. It just implements different types of simulators including ATARI games. Although here we will only focus on simple ones, such as the **Chain enviroment** illustrated below.

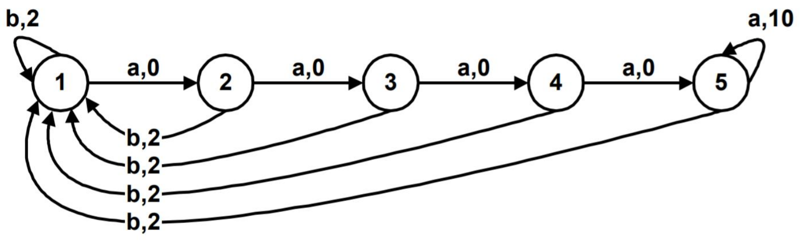

Figure 1. The "Chain" problem

The figure corresponds to an MDP with 5 states $S = \{1, 2, 3, 4, 5\}$ and two possible actions $A = \{a, b\}$ in each state. The arrows indicate the resulting transitions for each state-action pair, and the numbers correspond to the rewards for each transition.

Question 3 (optional)

You are to first familiarize with the framework of the OpenAI environments, and then implement the Q-learning algorithm for the NChain-v0 enviroment depicted above, using default parameters and a learning rate of $\gamma = 0.95$. Report the final Q^* table after convergence of the algorithm. For an example on how to do this, you can refer to the Q-learning of the **Frozen lake environment** (q_learning_frozen_lake.ipynb), uploaded on Canvas. Hint: start with a small learning rate.

Note that the NChain environment is not available among the standard environments, you need to load the gym_toytext package, in addition to the standard gym:

!pip install gym-legacy-toytext import gym import gym_toytext env = gym.make("NChain-v0")

```
[ ]: # Answer 3
```

Question 4

4a) What is the importance of exploration in reinforcement learning? Explain with an example.

Answer 4a: If we only exploit already known options, we can miss a better option which is not yet explored. For example we want to get from Augustów to Bydgoszcz as quickly as possible. Yet recently a new highway has been opened. We can try the new way and get to Bydgoszcz faster because there are many traffic lanes, or slower because there might be a traffic caused people who would like to try out the new road.

4b) Explain what makes reinforcement learning different from supervised learning tasks such as regression or classification.

Answer 4b: In supervised learning a model learns by example. It uses training data to predict output. Training data containg variables and labels. The model learns which values of variables give certain labels and applies it to new data. In reinforcement learning the learning agent is presented with an environment and must guess correct output. It explores the environment and gets an information how good the choice was, so that the Agent can iteratively learn to achieve a higher score.

Question 5

5a) Give a summary of how a decision tree works and how it extends to random forests.

Answer 5a: Using training data we build an optimal decision tree. In the tree there are nodes, which represent variables. When you want to know what should be a result for a new set of variables you go through decicion tree. You start at the root node. From each node you go to another node based on the value of the variable. At each step you go to the node with highest Information Gain and in the end you get the result at leaf node. We can extend decision tree to random forest. Using training data we build many different decision trees. Each decision tree works on a random subset of features to calculate the output. The random forest chooses the most often output of individual decision trees to generate the final output.

5b) State at least one advantage and one drawback with using random forests over decision trees.

Answer 5b:

Advantages:

- Reduced risk of overfitting - Decision tree is good for a sets of data similar to one on which model was trained. In a random forest the classifier won't overfit the model thanks to averaging of uncorrelated trees.
- Can handle both regression and classification tasks with a high degree of accuracy.

Disadvantages:

- Time consumimg - Needs time for calculations for each tree.
- Harder interpretation - It is more difficult to interpred the result of a random forest than a result of a decision tree.

References

Primer/text based on the following references: * http://www.cse.chalmers.se/~chrdimi/downloads/book.pdf * https://github.com/olethrosdc/ml-society-science/blob/master/notes.pdf