Assignment 5

May 2, 2023

0.1 Group 32: Saghar Alvandi, Sahar Abedi

0.1.1 Hours spent on the task: Saghar 15h, Sahar 15h

1 DAT405/DIT407 Introduction to Data Science and AI

1.1 2022-2023, Reading Period 4

1.2 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.

1.3 Primer

1.3.1 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**.

1.3.2 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.

1.3.3 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.

1.4 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 best way to navigate the MDP is by taking the EENNN path which gives a score of 0. Although there are other paths like EENNWNE, which have the same score, they are lengthier than the EENNN path. While the EENNN path is the most straightforward, it can be seen that there are different paths to reach the end with the same score. Therefore, the solution is not unique.

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:

Position	Optimal Action	Reward for Action
(0,0)	N/E	-1
(0,1)	E	1
(0,2)	N/W	-1
(1,0)	N/E	0
(1,1)	N/E/W/S	-1
(1,2)	N/S	1
(2,0)	N/E/S	-1
(2,1)	N/E	1
(2,2)	N	F
(3,0)	E	1
(3,1)	E	F
(3,2)	W/S	1

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

Answer 1c:

The expected total reward for a policy in an MDP is the sum of rewards obtained along the path when following that policy. The expected total reward for the policy of taking the EENNN path is 0, which is the maximum achievable reward in this MDP with a discount factor of 1. However, as mentioned in solution 1a, other paths also achieve a total reward of 0, so the optimal solution is not unique.

In our case, the policy of taking the EENNN path yields a reward of -1 when moving east from (0,0), a reward of 1 when moving east again from (1,0), a reward of -1 when moving north from (2,1), a reward of 1 when moving north from (2,1), and a reward of 0 when moving north from (2,2) to reach the goal state F. Therefore, the expected total reward for this policy is -1 +1 -1 + 1 = 0

1.5 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 ) +  *V_k-1[s ])
    end
    if  |V_k[s]-V_k-1[s]| < epsilon for all s
        for each state s,
        do
             (s)=argmax_a  _s  p(s |s,a)*(r(a,s,s ) +  *V_k-1[s ])
            return  , 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

1.6 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.

[14]:
```
import numpy as np

# Define reward and value matrices
r = np.array([[0, 0, 0], [0, 10, 0], [0, 0, 0]])
v = np.zeros((3, 3))

# Define constants
success = 0.8
discount = 0.9
epsilon = 1e-6

# Define function to get possible actions
def get_actions(x, y):
    # Define possible actions with their corresponding state transitions
    actions = [('N', (x - 1, y)), ('S', (x + 1, y)), ('E', (x, y + 1)), ('W',␣
↪(x, y - 1))]
```

```
    # Return only valid actions that don't go out of the grid
    return [(a, s) for a, s in actions if 0 <= s[0] <= 2 and 0 <= s[1] <= 2]

# Define value iteration function
def value_iter(r, v, success, discount, epsilon):
    while True:
        delta = 0
        for x in range(3):
            for y in range(3):
                old_v = v[x, y]
                actions = get_actions(x, y)
                expected_values = []
                # Calculate the expected value for each possible action
                for a, s in actions:
                    expected_value = success * (r[s] + discount * v[s]) + (1 -␣
 ↪success) * (r[x, y] + discount * v[x, y])
                    expected_values.append(expected_value)
                # Update the value function for the current state
                v[x, y] = max(expected_values)
                # Calculate the maximum change in value function for any state␣
 ↪in this iteration
                delta = max(delta, abs(old_v - v[x, y]))
        # Stop the loop if the maximum change in value function is smaller than␣
 ↪the threshold epsilon
        if delta < epsilon:
            break
    return v

# Call value iteration function
v = value_iter(r, v, success, discount, epsilon)
# Print the value function at convergence rounded to 2 decimal points
print('Converging optimal value function: ')
print()
print(np.round(v, 2))
```

```
Converging optimal value function:

[[45.61 51.95 45.61]
 [51.95 48.05 51.95]
 [45.61 51.95 45.61]]
```

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

Answer 2b:

The value iteration algorithm is designed to iteratively update the value function until it reaches its optimal form. This optimal form represents the maximum expected total reward that can be obtained, which starts from each state and follow the optimal policy. The convergence of the algorithm to the optimal value function is guaranteed as long as the discount factor is less than 1 and the threshold epsilon is small enough. It does not matter what initial value function we start with since the algorithm will eventually converge to the optimal value function. The reason for this is that the value iteration algorithm is designed to improve the value function in each iteration, and the optimal value function is the unique fixed point of the Bellman equation, which is the formula that updates the value function at each iteration of the algorithm.

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 represents the degree of importance given to future rewards as opposed to immediate rewards. It means that, a higher value of γ put more emphasis on long-term rewards, whereas a lower value of γ put more emphasis on short-term rewards.

When $\gamma = 0$, the agent only considers the immediate rewards and ignores all future rewards. This means that the agent does not care about the long-term consequences of its actions and only focuses on the present. As a result, the agent wants to choose actions that maximize immediate rewards, even if they lead to negative consequences in the future.

When $\gamma = 1$, the agent considers all future rewards and places equal importance on immediate and future rewards. This means that the agent cares about the long-term consequences of its actions and tries to maximize the total expected reward over the entire time. As a result, the agent wants to choose actions that lead to the highest expected total reward, even if they require ignoring some immediate rewards.

When deciding on which value of γ to use for a particular MDP, it is important to consider the time span of the problem, the type of rewards, and the computational complexity of the algorithm. A high value of γ may be suitable for problems with a long time horizon and where future rewards are significant, while a low value of γ may be good for problems with a short time horizon or where immediate rewards are more important. A high value of γ will make the algorithm more complex and will require more iterations to converge to the optimal solution.

1.7 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.

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.

1.8 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")

[9]:
```python
#!pip install gym-legacy-toytext

import gym
import gym_toytext
import numpy as np

env = gym.make("NChain-v0")

# Initialize Q-values table
Q = np.zeros((env.observation_space.n, env.action_space.n))

# Set hyperparameters
alpha = 0.1
gamma = 0.95
num_episodes = 1000
max_steps = 100

# Q-learning algorithm
for episode in range(num_episodes):
    state = env.reset()
    for step in range(max_steps):
        # Choose action based on epsilon-greedy
        if np.random.uniform() < 0.1:
            action = env.action_space.sample()
        else:
            action = np.argmax(Q[state])
        # Take action and observe next state and reward
        next_state, reward, done, _ = env.step(action)
        # Update Q-value table
        Q[state][action] += alpha * (reward + gamma * np.max(Q[next_state]) -␣
 ↪Q[state][action])
        # Move to next state
        state = next_state
        if done:
            break

print("Final Q-values table:\n")
print()
print(np.round(Q, 2))
```

```
Final Q-values table:


[[61.78 59.29]
 [64.49 59.98]
 [68.95 60.16]
 [72.83 59.02]
 [82.39 65.01]]
```

1.9 Question 4

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

Answer 4a:

Exploration is an essential aspect of reinforcement learning because it allows the agent to learn about the environment and discover the optimal policy for maximizing rewards. In RL, the agent learns by interacting with the environment and receiving feedback in the form of rewards. Through exploration, the agent can learn what actions result in the highest rewards and can refine its policy accordingly. Without exploration, the agent would simply repeat the same actions, never discovering new information or improving its performance. So, exploration is essential for an RL agent to learn and improve over time.

For example, as a student, I am trying to find the best study strategy for a difficult course. I could simply stick to the same study methods I've always used, but I might not be maximizing my potential. Alternatively, I could try out different study strategies, such as creating flashcards, summarizing lecture notes, or working through practice problems. By exploring these different strategies and analyzing their effectiveness, I can ultimately determine the most effective method for my learning style.

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

Answer 4b:

The main difference between reinforcement learning and supervised learning tasks such as regression or classification is the type of feedback provided to the agent. In supervised learning, the agent is given a labeled dataset and must learn to predict the correct output for new inputs. The agent is given explicit feedback on the correctness of its predictions so it can adjust its model based on that.

In reinforcement learning, the agent is not given labeled data. Instead, it learns from trial and error. The agent interacts with the environment and receives feedback in the form of rewards or punishments, based on the actions it choose. The agent should learn to relate its actions with outcomes and choose actions that maximize the expected cumulative reward over time.

Also, supervised learning normally involves a fixed dataset, whereas in reinforcement learning the agent is constantly learning and adapting to new situations. This process requires the agent to balance exploration and exploitation to learn about the environment and determine the optimal policy.

1.10 Question 5

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

Answer 5a:

A decision tree is a supervised learning algorithm that is used for classification and regression analysis. The algorithm works by partitioning the input data (hierarchical) into subsets, based on a set of splitting criteria. These criteria separate the target variable classes or values. The decision tree algorithm splits the data at each step (node of the tree) by choosing the best way to group the data based on a specific feature and value. The decision tree algorithm aims to make the subsets as similar as possible to each other and as different as possible from the other subsets. This process is repeated until the leaves of the tree contain pure or almost pure subsets of the target variable.

Random forest is an extension of decision trees that improves the accuracy and stability of predictions by combining multiple trees. The random forest algorithm creates many decision trees, each of them trained on a different random selection of the data and features. After making predictions with each tree, the random forest algorithm combines them by taking a vote or an average. Randomizing the data and features in the random forest algorithm prevents it from overfitting and creates a variety of trees. This leads to better performance when predicting new data.

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

Answer 5b:

One advantage of using random forests is they are less prone to overfitting and can handle noisy or high-dimensional data more efficiently, and also can generalize better to new data. Random forests also can provide a ranking of important features, which is useful for feature selection and interpretation. This ranking can help us identify the most important variables for predicting the target variable and improve the performance of the model, and gain insights into the relationships between input variables and the target variable.

The drawback of using random forests could be they can be computationally expensive and slow to train, especially for large or complex datasets. The reason is that they require multiple trees to be built and evaluated. Also, they can be less interpretable than decision trees which makes it difficult to understand the relationships between input features and the target variable. This is because in random forests the predictions are made based on the behavior of many trees together, rather than just one tree.