DAT565 Introduction to Data Science and AI

2023-2024, LP1

Assignment 5: Reinforcement Learning and Classification

Module 5 Group 32: Aghigh Merikhi(15 h) - Seyedehnaghmeh Mosaddeghi(15 h)

The exercise takes place in a notebook environment where you can chose to use Jupyter or Google Colabs. We recommend you use Google Colabs as it will facilitate remote group-work and makes the assignment less technical.

The exercise takes place in this notebook environment where you can chose to use Jupyter or Google Colabs. We recommend you use Google Colabs as it will facilitate remote group-work and makes the assignment less technical.

Tips:

- You can execute certain Linux shell commands by prefixing the command with a `!`.
- You can insert Markdown cells and code cells. The first you can use for documenting and explaining your results, the second you can use to write 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 (questions 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: $t = 1, 2, \dots, T$, where $T \leq \infty$
- State space: $S = \{s_1, s_2, \dots, s_N\}$ of the underlying environment
- Action space $A = \{a_1, a_2, \dots, a_K\}$ 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,

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

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. For instance, NESW will make a circle.

The rewards associated with transitioning between states are as follows:

- Moving (N) or (S) results in a reward of -1.
- Moving (E) results in a reward of 1.
- Moving (W) results in a reward of 0. The goal is to find the optimal path that leads from 'S' to 'F' while maximizing the cumulative reward. One possible optimal path is 'EENNN'. The cumulative reward of -1 + 1+(-1) + 1 = 0 indicates that this path is optimal in terms of maximizing the total reward. Paths like EENNWNE and is valid and result in a score of 0. Thus, the optimal path in the MDP can have multiple valid solutions with the same score.

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.

Position	Optomal Action	Reward for action
(0,0) = **S**	N/E	-1
(0,1)	E/N	0
(0,2)	N/E/S	-1
(0,3)	E	1
(1,0)	E	1
(1,1)	N/E/W/S	-1
(1,2)	N/E	1
(1,3)	E	F
(2,0)	N/W	-1
(2,1)	N/S	1
(2,2)	N	F

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

As explained above, the expected total reward for the optimal policy 'EENNN' is 0. This indicates that the agent successfully navigates from 'S' to 'F' while maximizing its cumulative reward. This path results in the agent navigating through the MDP and accumulating rewards as it moves, ultimately reaching the goal state 'F' with a cumulative reward of 0. Furthermore, other paths can also achieve a total reward of 0 in this MDP, making the optimal solution not unique. The MDP allows for multiple valid paths to reach the same maximum achievable reward.

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_k-1[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_k-1[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.

In [ ]:
```
import numpy as np

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

# Define constants
action_probability = 0.8
discount_factor = 0.9
epsilon = 1e-4

while True:
    delta = 0
    for x in range(3):
        for y in range(3):
            old_v = value[x, y]
            actions = [('N', (x - 1, y)), ('S', (x + 1, y)), ('E', (x, y + 1)), ('W
            expected_values = []

            for a, s in actions:
                if 0 <= s[0] <= 2 and 0 <= s[1] <= 2:
                    expected_value = action_probability * (reward[s] + discount_fac
                    expected_values.append(expected_value)
                else:
                    expected_value = action_probability * (reward[x, y] + discount_
                    expected_values.append(expected_value)
```

```
            value[x, y] = max(expected_values)
            delta = max(delta, abs(old_v - value[x, y]))

    if delta < epsilon:
        break

# Print the value function at convergence
print('Converging optimal value function: ')
print()
print(np.round(value, 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 V0.

In value iteration, the convergence criterion is based on the maximum change in the value function (that is, when the change goes below a specific threshold, represented as epsilon). This criterion ensures that the algorithm ends when the values have stabilized, resulting in the optimal solution. The convergence qualities of the method ensure that the result is independent of the initial values. While the initial values may affect the amount of iterations needed to converge, the ultimate solution is always the same—the optimal policy and value function for the particular MDP.

2c)Describe your interpretation of the discount factor gamma. 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 gamma to use?

The discount factor gamma determines the trade-off between immediate and future rewards. A higher gamma values indicate that the agent values long-term rewards more and is willing to delay immediate reward, whereas a lower gamma values indicate a preference for immediate rewards. Below are the two extreme cases:

γ = 0: The agent only considers and focuses on maximizing immediate rewards, and it ignores future rewards nd long-term effect completely. In many real-world settings, the agent's purpose is to maximize immediate rewards, which may not be the optimum strategy.

γ = 1: The agent has a long-term perspective and considers future rewards as highly as immediate rewards in this extreme scenario. The agent takes into account the cumulative expected rewards over time and tries to maximize the overall expected reward throughout the entire Track. The agent is willing to postpone benefits in order to get larger rewards later.

Choosing the appropriate discount factor value in a MDP is a vital decision that must balance short-term and long-term goals. Risk tolerance, time horizon, uncertainty, computing efficiency, and the discounting of distant rewards should all be variables in the

selection. Experimentation and sensitivity analysis are frequently used to determine the ideal value that matches with the unique problem's dynamics and the agent's aims, while knowing that it fluctuates depending on the characteristics and goals of the situation.

Question 4

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

Exploration is an important concept in reinforcement learning (RL) because it enables an agent to learn about its environment and discover optimal strategies. In RL, an agent often starts with limited knowledge of the environment and must explore different actions and states to gather information. Without exploration, the agent might miss valuable rewards, get stuck in repeated actions, or never discover hidden opportunities. Balancing exploration and exploitation is essential for agents to learn and make optimal decisions in uncertain environments. For example, an online recommendation system that suggests movies to users. The system begins with no prior knowledge of users' preferences and must learn to make accurate movie recommendations. Exploration in this context involves recommending a variety of movies to different users, even if the system is uncertain about their preferences. And exploration allows the recommendation system to learn about users' movie preferences over time.

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

In RL, agents tries to make decisions over time to maximize a cumulative reward, with feedback provided in the form of rewards based on their actions. In contrast, supervised learning involves training a model to make predictions or classifications based on labeled data, without interactions with an environment. RL operates in environments where labeled datasets are not available, it learns from exploration and trial and error, whereas supervised learning relies on available labeled data for training. RL deals with feedback delays, as rewards may be delayed in time and attributed to a sequence of actions, whereas supervised learning typically receives immediate and direct feedback for each prediction.

Question 5

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

A decision tree is a supervised machine learning approach that divides a dataset into subsets recursively according to feature values. Each decision node represents a characteristic and a splitting criterion, forming the decision nodes and leaf nodes of a tree-like structure. The method chooses the optimum features and split points to optimize a certain criteria, such as mean squared error reduction for regression or Gini impurity for classification. Data points

follow the splits in the tree based on their feature values as they move from the root to a leaf node to create predictions. The final output is the prediction made at the leaf node.

By assembling several different decision trees, Random Forests expand the idea of decision trees. Each tree examines just a random subset of characteristics at each node split and is trained on a random subset of the training data with replacement. In Random Forests, the final prediction is obtained by combining the predictions of individual trees, either by majority voting for classification problems or averaging for regression tasks.

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

Random Forests have the following advantage over Decision Trees:

Reduced Overfitting: When compared to individual decision trees, Random Forests are less prone to overfitting. They develop a more generic model by averaging predictions from many trees and employing random feature selection, which generally results in higher performance on unknown data. As a result, Random Forests are more robust and suitable for a broader range of datasets.

The disadvantage of Random Forests over Decision Trees is as follows:

Random Forests are fundamentally more complex than individual decision trees since they are made up of several trees. This intricacy can make interpreting the model's predictions and understanding the significance of specific variables more difficult. A single decision tree, on the other hand, is usually easier to depict and describe.

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