DAT405/DIT407 Introduction to Data Science and AI

2022-2023, Reading Period 4

Assignment 5: Reinforcement learning and classification

Authors: Kevin To and Filip Cederqvist

Work load: 11 h each

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: $t = 1, 2, \ldots, T$, where $T \leq \infty$
- State space: $S = \{s_1, s_2, \ldots, s_N\}$ of the underlying environment
- Action space $A = \{a_1, a_2, \ldots, a_K\}$ 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 $\pi : s \rightarrow a$, 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. E.g. NESW will make a circle.

Answer 1a: The optimal, and unique, path is EENN, seeing as it is the only path that gives a non-negative reward (-1+1-1+1 = 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: The optimal policy is to move to the square that gives the highest reward, given that you can't move back to the square you came from. If two squares give the same reward, you should consider the square that gives the highest reward after in the next step, and so on.

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

Answer 1c: The expected total reward is 0, since the policy in 1a) gives a reward of 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_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. Make sure to consider that there may be several equally good actions for a state when presenting the optimal policy.

In [9]:
```
# Answer 2a
import numpy as np
epsilon = 1e-8
gamma = 0.9
action = ['N', 'E', 'S', 'W']

# define a function to reture the cordinates for the next state and the name
def possible_moves(x, y, dim):
# Checks all possible ways we can move in the grid depending on the position
    (m, n) = dim
    moves = []
    if y > 0:
        moves.append([x, y - 1, 'N'])
    if y < m - 1:
        moves.append([x, y + 1, 'S'])
    if x < n - 1:
        moves.append([x + 1, y, 'E'])
    if x > 0:
        moves.append([x - 1, y, 'W'])

    return moves

# defining the value interation function
def value_iteration(rewards, epsilon, gamma):
    # We initialize the value function and the last value function to zero.
    (r,c) = rewards.shape
    Value = np.zeros((r, c))
    Last_value = Value.copy()
    probability_move = 0.8
    policy = np.zeros((r, c), dtype=str)
    iterations = 1
    # We run the loop until the value function converges.
    while True:
        for row in range(r):
            for col in range(c):
                moves = possible_moves(col, row, rewards.shape)
                # Calculating the value for all possible actions and choosin
                for x, y, a in moves:
```

```
                    V_calculated = probability_move*(rewards[y, x] + gamma*L
                    if V_calculated > Value[row, col]:
                        Value[row, col] = V_calculated
        # Check if the value function converges, if it does we return the op
        if (np.abs(Value-Last_value) < epsilon).all():
            for row in range(r):
                for col in range(c):
                    moves = possible_moves(col, row, rewards.shape)
                    current_largest = probability_move*(rewards[moves[0][1],
                    # comparing the values of all possible actions and retur
                    for x, y, a in moves:
                        V_calculated = probability_move*(rewards[y, x] + gam
                        if V_calculated >= current_largest:
                            policy[row, col] = a
            return Value, policy, iterations
        # after each iteration, we update the value function. And add to the
        Last_value = Value.copy()
        iterations += 1

rewards = np.zeros((3, 3))
rewards[1][1] = 10
Value, policy, i = value_iteration(rewards, epsilon, gamma)
print('\nThe optimal value function for each state:')
print(Value)
print('\nThe optimal policy for each state:')
print(policy)
print('\nThe number of iterations it took to converge:', i)
```

```
The optimal value function for each state:
[[45.61292358 51.94805187 45.61292358]
 [51.94805187 48.05194797 51.94805187]
 [45.61292358 51.94805187 45.61292358]]

The optimal policy for each state:
[['E' 'S' 'W']
 ['E' 'W' 'W']
 ['E' 'N' 'W']]

The number of iterations it took to converge: 192
```

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

Answer 2b: The result does not depend on the initial value because the value iterations will converge to the same value no matter what the initial value is. Setting the initial value to 0 is just a convention. Setting it to some other value would just be an approximation of the optimal value function. The optimal policy will be the same no matter what the initial value is. However, if you can make an approximation of the optimal value function, you could make it converge faster

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: γ is the discounting factor and determines how future values are weighted. If you have γ = 0, you only take the next move into account, so future rewards are discarded. If you have γ = 1 you weight all rewards the same, e.g if you get the reward now or in ten moves does not matter. If you only have a few move to make, you should have a low γ, but if you don't care about "short term" success you could probably use a higher γ.

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.

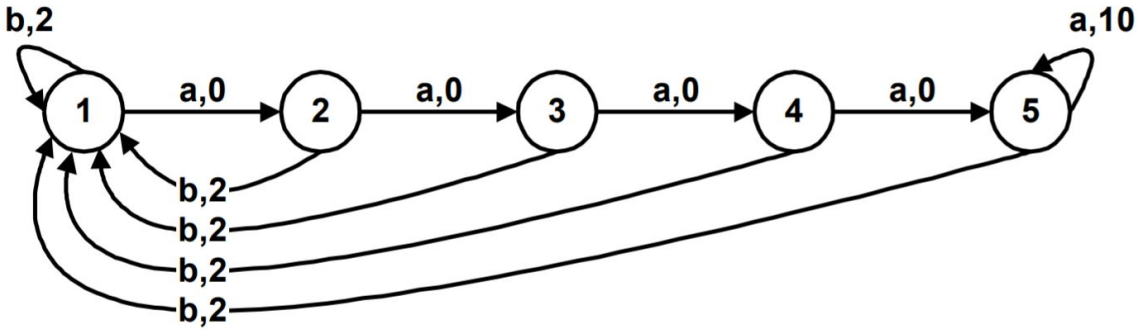

Figure 1. The "Chain" problem

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")
```

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

Question 4

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

Answer 4a: Exploration is important in reinforcement learning because it allows the agent to try new actions that would otherwise be missed. If the agent only exploits the policy it might not find the optimal policy. This can be illustrated with an example from the real world: you are in a new city and you want to find the best restaurant. If you only exploit the policy you will only go to the restaurant you know is good. However if you were to explore you might find a even better restaurant!

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

Answer 4b: The difference between reinforcement learning and supervised learning is that in reinforcement learning you don't have a set of training data. In reinforcement learning you have to explore the environment and learn from the rewards you get. In supervised learning you have a set of labeled training data which the model find patterns and learn from.

Another difference is in the way the models learn. In supervised learning, the model is given direct and explicit feedback on its performance, which it learns from. In reinforcement learning, however, the model is given indirect feedback on its performance via rewards and punishments.

Question 5

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

Answer 5a: A decision tree uses a tree-like model of decisions and their possible consequences. Each node in the tree represents a decision or a test on a feature, and each branch represents the possible outcomes of that test. The leaves of the tree represent the predicted outputs or classifications.

To build a decision tree, the algorithm recursively splits the input data into smaller subsets based on the input features that best separate the classes or minimize the variance of the target variable. The goal is to find the most informative features and splits that maximize the information gain or Gini impurity, which measures the purity or homogeneity of the resulting subsets.

However, decision trees have some limitations, such as overfitting and sensitivity to the choice of input features and splitting criteria. Due to this, random forests was developed. These are a collection of decision trees, each trained on a different subset of the input features and training data. The final prediction is obtained by averaging or voting over the predictions of all the trees.

Random forests help to reduce overfitting and improve the generalization performance of decision trees. They also provide feature importance measures, which can help identify the most relevant features for the classification or regression task.

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

Answer 5b: One advantage of using random forest over decision trees is that it doesn't suffer from the overfitting problem. This is because random forest takes the average of all the predictions, which cancels out the biases. However, random forests are more complex and take longer to make predictions compared to decision trees. Random forest are also more difficult to interpret.

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