Programming Assignment: Reinforcement Learning

Deadline
For grade **A** you must have your implementations accepted by Kattis with the required scores by the December 22, 2021, 5pm sharp and presented within the regular examination period. Any assignment accepted by Kattis/presented after these deadlines will result in max. grade B.

Rules
You should work in pairs, and you can discuss and divide the work within your pair, as you wish. However, you will be assessed individually. This means that both people have to be able to explain the theory as well as the details of the implementation.
You can work individually, but you cannot work in groups larger than two people.

Introduction - FishingDerbyRL MDP (Markov Decision Process)

In this assignment you will delve into a new formulation of the previous fishing problem, the FishingDerbyRL environment, to learn about Reinforcement Learning (RL). You will learn about the environment and implement an agent (a diver) to be proficient at catching the King Fish.

A graphical representation of the FishingDerbyRL environment is illustrated in Fig. 1. Your agent will control the diver (in green) by diving (left, right, up and down) to catch the orange King Fish (Fig. 2b), while avoiding the Jelly Fish (Fig. 2a). The FishingDerbyRL environment can be characterized as an MDP (Markov Decision Process) and assumes:

- **State Space** $\mathcal{S}$: All possible states of the environment explorable by the agent.
- **Action Space** $\mathcal{A}$: All possible actions the agent can perform at each state.
- **Transition Function** $\mathcal{T}$: The probability distribution of reaching all possible next states given the previous state and an action.
- **Reward function** $\mathcal{R}$: The reward obtained by the agent when arriving at state s_{t+1} by performing action a_t at state s_t.

The reward function in this domain is represented by a sum of three terms:
$R = R_{living} + R_{jelly} + R_{king}$:

- **Living Reward** R_{living}: A zero or small negative reward by each step (when transitioning from s_t to s_{t+1}).
- **Jellyfish Reward** R_{jelly}: A negative reward obtained when colliding with a jelly fish (see Fig 2a).
- **King fish Reward** R_{king}: A positive reward obtained when catching the king fish (see Fig 2b).

Figure 1: Graphical representation of the environment.

(a) Jellyfish

(b) King Fish

Figure 2: Fish species in the FishingDerbyRL Environment

Getting started

In this assignment, you are required to understand and implement different strategies for training reinforcement learning agents. Along with the description of your tasks, we pose a number of questions, marked by a frame in the text. You are expected to be able to answer these questions in the presentation session.

For grades E and D, you need to implement solutions to all problems given in sections 1 and 2 and pass the respective points mentioned in the exercise. **For grade C**, you need to have passed the E-D level and you are furthermore required to complete exercises 3 and 4 with the corresponding points mentioned in the exercises. Finally, **for grades A and B**, you need to have passed the E, D and C levels and solve the assignment given in section 5. In the final assignment, you are required to apply your knowledge to a slightly more difficult problem. You will need to escape a sub-optimal policy to find the optimal policy across different unseen scenarios.

Important note: You're strongly encouraged not to use the built-in **random** module in any of the problems in this lab. The reason behind this is that it will give you unreliable results. Instead, use **np.random**!

1 Grade Level E and D

1.1 An overview of the environment

In this part of the assignment, you are getting familiar with the environment by implementing a simple random agent. You will make the first steps towards setting up a loop with your agent and the environment which will be useful in the following exercises. Download the package **rl1.zip** from kattis RL-1 and extract the contents. The folder contains several files, but for now focus on the **app_manager.py** file which defines the environment implementation. Observe the **FishingDerbyRLApp** class and in particular the **init_states(...)**, **init_actions(...)** and **step(...)** methods.

Define the following properties of the FishingDerbyRL MDP:

- **State Space** $\mathcal{S}$: The total number of states of the environment (which is represented by a grid world).
- **Action Space** $\mathcal{A}$: All possible actions carried by the diver.

1.2 Random agents

Every agent (reactive, random or learning) requires a communication loop as expressed in Fig 3. A Markov Decision Process (MDP) is unrolled in a series of decision making sequences. Each episode comprises a series of steps. In the FishingDerbyRL, each episode terminates after 100 steps or when the diver catches the King Fish (Fig 2b).

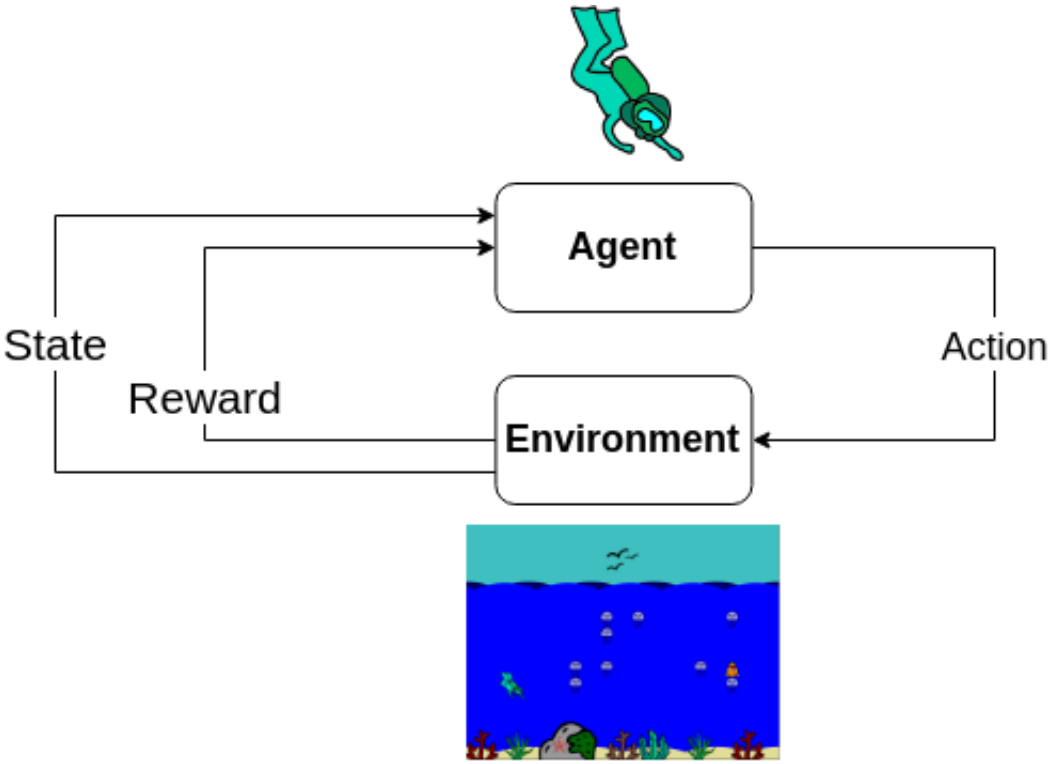

Figure 3: Illustration of the reinforcement learning loop. The agent (diver) acts upon the current state of the environment (sea), and transitions to the next state with a reward.

Within the folder from the package **rl1.zip** you will edit the **player.py** file. The file contains several base classes with controllers where you will implement your agents in this assignment. In order to advance to implementing a learning algorithm, you need firstly to be able to establish a baseline of interaction between the environment and the agent. To achieve this, you will implement your first agent, a random agent. Your agent will need to perform the following:

- Sample a random action from the allowed action space $\mathcal{A}$
- Check that the sampled action is sent to the environment.
- Observe the stopping criteria and change the number of running episodes.

To perform the task, look for a **PlayerControllerRandom** class within **player.py**. You will edit the **self.random_agent()** method within the class. Your agent will interact randomly with the environment by choosing a random action at each state. The final policy returned by your agent will be the most picked random action at each state, across all episodes. Commented inline, you will find the place to insert your code. Open and observe **settings.yml** – this file defines the MDP in a higher level. The flag **headless** is *true* by default, in order to improve the speed of the simulation and omit the graphical interface. Set the flag to *false* to observe your agent's actions in real time.

When you finish, you can test your random agent by executing: **"python main.py settings.yml"**. You should be able to see messages printed on the standard output (see Figure 4).

```
Episode: 0, Steps 100, Diff: 4.042803e-01, Total Reward: -163, Total Steps 0
Episode: 1, Steps 100, Diff: 4.595717e-01, Total Reward: -172, Total Steps 100
Episode: 2, Steps 100, Diff: 4.649030e-01, Total Reward: -208, Total Steps 200
Episode: 3, Steps 98, Diff: 2.971271e-01, Total Reward: -167, Total Steps 300
```

Figure 4: Example of standard output episode information.

How to submit: Edit **player.py**, rename it as **player_1.py** and upload it to Kattis RL-1. To pass you need to acheive 10 points. **Tip**: You can always find where to add your code snippet just by entering the exercise index. For example, just search for "1.2" within **player.py** to find your code placement.

2 Grade Level E and D- Q-learning

In this exercise, you will implement a Q-learning agent and dive deep in the FishingDerbyRL environment. Your resulting agent will be evaluated through a series of scenarios.

In model free reinforcement learning, your agent does not have access to the dynamics of the full MDP *a priori*. Thus, your agent will learn by interacting directly with the environment and, consequently, gathering rewards, i.e, **learning by reinforcement**. The goal of your agent is to maximize the sum of all discounted rewards gathered by your agent until the end of each episode. Q-learning is a model-free, active reinforcement learning algorithm.

2.1 Initialization for Q-Learning

Within the folder from the package **rl2.zip** you will continue editing the **player.py** file. The file contains several more base classes with controllers. You will inspect the **PlayerControllerRL** class and add your code in the space commented inline with further instructions. We recommend to work step by step and first do the following:

- Inspect the **self.q_learning()** method and initialize a Q-table with the correct size.
- Repeat the steps of the previous exercise of your random agent.

In this exercise use the file **settings.yml** by also running "**python main.py settings.yml**". To run the RL agent, change the value of *player_type* to *"ai_rl"* (you can do this by changing the comment on the lines). Your agent should be able to run until **self.episode_max** editable in **settings.yml**

Pro Tip: Do not hardcode parameters of your Q-learning algorithm. Instead, edit them directly in **settings.yml**

2.2 Implement the Q-value update

As a next step, you will implement the *Q-learning update equation* to update your Q-table initially instantiated above. More specifically:

- Compute the *Q-learning update* by using the correct learning rate **alpha (α)** and discount factor **gamma (γ)** available in the **settings.yml**.
- Test and visualize (do not forget the *headless* flag on **settings.yml**) the performance of your agent.
- Change the exploration strategy from random to choosing the action with the highest *Q-value* for the current state.

What do you observe? Is your agent capable of achieving the maximum score of 11 (if not, please explain why). Test several times using the same amount of episodes. Whenever you start each run, a random seed is generated. Feel free to uncomment the seed parameter and set a specific seed in **settings.yml**.

2.3 Convergence of training

Until now, you used a simple convergence criteria, depending solely on the maximum number of episodes. However, there are other common convergence criteria in the literature. As your agent interacts and learns in the environment, its predictions on the value and quality of its actions within every state become more accurate, and the update error becomes lower.

- Keep track of the last update error at each step, i.e, the difference between the predicted Q-value and the actual Q-value for the current state.
- Implement the convergence criteria of the algorithm to not only include the maximum allowed number of episodes, but also a **threshold** of 10^{-6} provided as property of the class in **self.q_learning()** (can be edited through settings.yml) method and compare it with the previous mean difference of the Q-value table.

How to submit: Edit **player.py**, rename it as **player_2.py** and upload it to Kattis RL-2. To achieve a grade of E you need to obtain at least 25 points, and for a grade of D at least 35 points.

3 Grade C - Exploration vs Exploitation

In this exercise, you will delve into the Exploration vs. Exploitation dilemma. A static and a dynamic exploration strategy will be implemented and used in your agent from the previous exercise. One dichotomy in reinforcement learning relates to the paradigm of exploration vs exploitation. While interacting with the environment, the agent (our learning algorithm) may choose to explore or exploit at each step. Both impact learning differently, and a balanced strategy is required. Most approaches follow the trend of initially maximizing exploration to interact with the environment as much as possible, and exploit once the most promising paths have been explored. In many recent exploration strategies, the objective deals with maximizing the entropy of state observations rather than randomly exploring. Currently, there are many methods available to schedule exploration rates, depending on how action selection is modelled. The classic exploration method *epsilon-greedy* is described below. However, there are many other strategies such as Boltzmann exploration and maximum entropy exploration.

3.1 Epsilon-greedy algorithm

So far, in the previous exercises, you implemented two exploration strategies. In the first exercise you implemented a random agent, and in the second exercise you implemented a *greedy* agent, i.e, an agent that always chooses the highest perceived valued action. The first will very rarely achieve the optimal strategy, unless the optimal policy is indeed a random one. The latter (*greedy* strategy) can often lead to sub-optimal policies, by never exploring policies with low short-term gains but rather larger long-term pay-offs.

You will delve into the problem of *exploration vs exploitation* by implementing a hybrid method, which simultaneously explores and exploits: the epsilon-greedy algorithm.

The epsilon greedy method comprises on selecting actions greedily with probability $(1 - \epsilon)$ (thus the name epsilon-greedy). Selecting a greedy action is equivalent to choosing the action with the highest Q-value, i.e., $argmax_{a_t} Q(s_t, a_t)$. Conversely, actions are selected randomly with probability *epsilon* $(0 < \epsilon \leq 1)$.

To complete this task, consider the **player.py** file and look at the **PlayerControllerRL** class. Find and edit the **self.epsilon_greedy()** method and implement the epsilon greedy algorithm with the given default arguments. Change your exploration strategy implemented previously in the Q-learning exercise and test your agent. You will:

- Implement the epsilon-greedy algorithm;
- Add it to the action selection phase of your Q-learning agent.

Pro Tip: Do not hardcode parameters of your epsilon-greedy algorithm, edit them directly in **settings.yml**

3.2 Annealing

Another approach would consider a higher exploration phase when the environment is unknown to our agent, and an exploitation phase when the agent is more certain of the value of its actions. Thus, trade-off in exploration vs exploitation is made by varying (annealing) ϵ. An ϵ closer to 1 relies on pure exploration (always selecting random actions) while a value closer to zero, maximizes exploitation by always relying on the highest Q-value for each state. A scheduling function may then be constructed to regulate the value of ϵ. One option is to construct a linear schedule for ϵ in the following manner:

Your task consists in finding the class **ScheduleLinear** in the **player.py** file and completing the **value()** method. Thereafter, change your Q-learning algorithm to include the annealing of epsilon within your epsilon-greedy algorithm:

Input: $\epsilon_{initial}$, ϵ_{final}, timestep t and total timesteps T, state s_t action a_t and action space $\mathcal{A}$;
$\Delta\epsilon = \epsilon_{final}$ - $\epsilon_{initial}$;
$\epsilon_t = \epsilon_{initial} + \Delta\epsilon \cdot (\frac{t}{T})$;
if *random(0,1)* $< \epsilon_t$ **then**
  return random a_t from $\mathcal{A}$;
else
  return $argmax_{a_t} Q(s_t, a_t)$;
end

Algorithm 1: Epsilon-greedy with linear schedule.

- Build a scheduling method to anneal the value of ϵ from 1.0 to 0.2 during 10000 steps by default.
- Integrate the implemented scheduling method with the action selection phase of your Q-learning agent.

How to submit: Download **rl3.zip** and edit **player.py**, rename it as **player_3.py** and upload it to Kattis RL-3. To pass you need to obtain at least 25 points.

4 Grade Level C - Hyperparameters and Environment Dynamics

In this part of the assignment, you are expected to adjust parameters in order to enhance the performance of your agent and prove your understanding both on the implementation of the algorithm and the dynamics of the environment.

An important aspect of the Q-learning algorithm is the learning rate α ($0 < \alpha \leq 1$), which controls the size of the update at each learning step: a value closer to 0 leads to a slow update, thus learning, while a value closer to 1 leads to high variance. γ represents the discount factor ($0 < \gamma \leq 1$). A low γ value characterizes a greedy agent, over-valuing instant rewards and devaluing long-term rewards.

Both hyperparameters α and γ depend on the genesis of the problem. There are many algorithms focusing on the optimization of this hyperparameters, under the scope of *meta-learning*.

4.1 Hyperparameters

The above hyperparameters refer to the Q-learning algorithm. Define and test at least one interval for each of them accordingly, in order to achieve the following desirable policies:

- **(1)** Not improving/learning.
- **(2)** High variance but fast learning.
- **(3)** Low variance and high long-term return.
- **(4)** High variance and high long-term return.

4.2 Dynamic environments

In this part of the exercise you will change several Q-learning hyperparameteres and the environment MDP simultaneously. Both will influence the optimal policy of your agent. Take a look at the files **student_4_2_1.py** and **student_4_2_2.py**. The first line comprises the rewards of the MDP, as commented in the file. The reward function is represented by a sum of three terms $R = R_{living} + R_{jelly} + R_{king}$, which were explained in the introduction. The rewards are comprised as a list, where:

- The **first element (rewards[0])** corresponds to the reward for catching the King fish R_{king}
- The **following elements (rewards[1:n-1])** correspond to the reward for catching a Jelly fish R_{jelly}
- The **last element (rewards[n])** corresponds to the living reward (reward per step) R_{living}.

The other hyperparameters refer to the hyperparameters of Q-learning and epsilon-greedy. All numeric values can be filled either as floats or integers. Edit the parameters accordingly and test them to reach the desirable behaviours:

- **(1)** Never catching any Fish, including the King Fish.
- **(2)** Taking the shortest path to catch the King Fish.

You are able to visually check the effect of these changes on your own by editing **settings.yml** and using your previously obtained agent and setting the headless flag to false.

The optimal policy is achieved when the expected long-term return is maximized. More generally, the computation of optimal policies in complex MDPs requires a large number of training episodes.

> However, if the reward structure of an MDP is simple enough, the optimal policy degenerates in a simple heuristic. Given the **4_2_3.yml** reward structure and initial position of jelly fish/king fish/diver, what is the value of the long term return of the optimal policy?

How to submit: Download **rl4.zip** and adjust the parameters in the files **student_4_2_1.py** and **student_4_2_2.py**, once you have tried them in the **settings.yml**. Change only the values without changing the variable hyperparameter names. Use the previous **player.py** (or another one to run your parameters) and rename it as **player_4.py**. All files should be submitted to Kattis RL-4. To pass you need to obtain at least 10 points.

5 Grade Level B and A - ESCAPING a sub-optimal policy

While your agent is exploring and learning, it might arrive at a *sub-optimal policy*, i.e. a policy with a lower long-term return than the optimal (global) policy. This can happen for a variety of reasons, for example:

- MDP dynamics, specifically in large state spaces;
- Randomness inherent to exploration;
- Hyperparameters of the learning algorithm and convergence criteria.

5.1 Strategies for learning

This is the final test (boss) for your Q-learning agent. Use the settings provided in **5_student.yml** and transfer them to the settings.yml to run your agent. Test your agent several times. Is your agent always able to reach the same policy? You will need to be able to use what you have learned and what was stated above to be able to reach the optimal policy, regardless of randomness. You can improve and change your agent in several ways:

- Using the best exploration strategy you learned;
- Chose the optimal hyperparameters for learning;
- Adapting your convergence criteria.

Test your agent several times to ensure that it always reaches the optimal policy. For the default reward structure in **settings.yml** file, a policy that always reaches a total reward of 11, with different seeds, is optimal. Your agent will be tested in three scenarios of similar complexity.

How to submit: Download **rl5.zip**, complete and submit the files **student_5.py** with your parameters (insert only the values for the corresponding variables, other parameters will not be read). Edit **player.py** and rename it to **player_5.py**, then upload it to Kattis RL-5.

Tip: Keep track of the highest return $\mathcal{R}$ per episode for the last N episodes. If the values are similar and converge to the highest return found, convergence is near (which can be used alongside the threshold used in exercise 2.3). Always return the best policy. Annealing other hyperparameters such as the learning rate α may help. Assume your agent can train a maximum of 2000 episodes (each having a maximum of 100 steps) and use the **self.episode_max** as given to you in **player.py**. A score of 0 will be given if the agent uses more than the maximum number of episodes and steps. Test on your side with different seeds to check the robustness of your solution.

Scoring for each scenario: Your score will depend on how your agent is performing in regard to three different criteria:

1) The obtained policy should always catch the King Fish (4 points);
2) If 1) is met, the diver should always do it in the optimal way (optimal policy) (2 points);
3) As stated above it's possible for your agent to train for 2000 episodes. However, if your agent needs more than 750 episodes to train, your score will be decremented. Furthermore, the fewer amount of episodes the agent needs, the better score you receive (-6 – 8 points).

The final score will be the sum of the three scenarios tested on Kattis. **To receive grade A you need to obtain at least 26 points before the deadline.**