Reinforcement Learning KU (708.062) WS23

Assignment 2

Policy Iteration & Model-free Reinforcement Learning

Assigned on:	24.11.2023 12:30
Q&A Session:	01.12.2023 12:30
Deadline:	**21.12.2023 23:55**
Submission:	Submit your report (pdf) and Python code files (solved code skeleton). via TeachCenter: https://tc.tugraz.at/main/course/view.php?id=3110
Group size:	Groups of two students are allowed for this task.

General remarks

Your submission will be graded based on:

- Clarity and correctness (Is your code doing what it should be doing?)
- Include intermediary steps, and textually explain your reasoning process.
- Quality of your plots (Is everything clearly legible/visible? Are axes labeled? ...)
- Both the submitted report and the code. If some results or plots are not described in the report, they will **not** count towards your points.
- Your submission should run with **Python 3.8+** and **gymnasium version 0.29.1**.
- Code in comments will not be executed or graded. Make sure that your code runs.

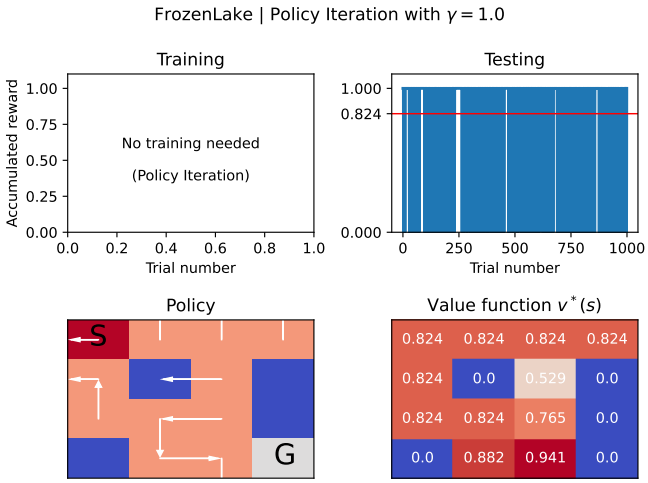
1 RL in a grid world [5 Points]

A grid world is a typical environment with finite action and state spaces. We will solve the FrozenLake[1] environment from the `gymnasium` package (a fork of OpenAI's Gym package). The agent controls the movement of a character in a grid world. Some tiles of the grid are walkable, and others lead to the agent falling into the water. Additionally, the movement direction of the agent is uncertain and only partially depends on the chosen direction (the lake is slippery). The agent receives a reward of 1 if it moves the goal state and 0 else. The episode ends as soon as the agent falls into the water or reaches the goal state.

a) Assume we know the dynamics of the underlying MDP (i.e., the state transition probabilities and the expected rewards). Fill in the TODOs in `ex1_policy_iteration.py`: You are supposed to implement the *Policy Iteration* algorithm, which consists of repeatedly *evaluating a policy* (using *Iterative Policy Evaluation*) and improving this policy by acting greedily w.r.t. the Q function of the old policy. When finished with the TODOs, execute the file: This will run policy iteration until convergence for both the undiscounted case ($\gamma = 1$) and a discounted case ($\gamma = 0.95$). In both scenarios, the code framework will produce a plot which includes (1) the policy you have found, (2) the corresponding value

[1] https://gymnasium.farama.org/environments/toy_text/frozen_lake/

(a) FrozenLake environment. The starting state is on the top left while the goal state is on the bottom right.



(b) Value function and policy that were acquired using *Policy Iteration* ($\gamma = 1$). Testing the policy using 1000 episodes yields an average sum of rewards of 0.824.

function, and (3) the reward that was accumulated in 1000 simulated episodes (and its average). Include both plots ($\gamma = 1$ and $\gamma = 0.95$) in your report. Explain any differences between the two policies, the two value functions, and between the average test success rate (i.e., the fraction of simulated episodes in which the agent reached the goal state). Briefly discuss why these differences appear.

b) Assume that s_{start} is the starting state. In the undiscounted case ($\gamma = 1$), what is the relationship between $v^*(s_{\text{start}})$ and the average reward you encounter when acting according to π^* for N episodes (i.e., `np.mean(policy_iteration.test_policy(num_episodes=N))`)? What happens as $N \to \infty$? Write down the definition of the v function in this particular case ($\gamma = 1$, binary reward per episode).

c) In practice, the MDP dynamics are often unknown in practice. Hence, we will now implement algorithms that *do not rely on knowing the MDP dynamics*. Carefully read the code in the file `ex1_model_free_rl.py` and fill in the TODOs. More specifically, this requires you to implement the following algorithms, which differ in how they update their current estimate of the Q values:

- SARSA (State, Action, Reward, State, Action)
- Q-Learning
- Expected SARSA

Moreover, you are tasked to implement an ε-greedy policy. Start training your agent with $\varepsilon = 1$ and decay ε throughout training using the `eps_decay` parameter. Train your agent for $10,000$ episodes and test the policy you have found using 5000 episodes. **Important:** When simulating test episodes, choose all actions deterministically ($\varepsilon = 0$).

For each of the 3 implemented algorithms and for each $\gamma \in \{0.95, 1\}$, find a good value for the hyperparameters α ("learning rate") and `eps_decay`. For each of the 6 cases, report the best hyperparameters you have found and include the plot that is saved by `plot_frozenlake_model_free_results`. Moreover, report which values of α and `eps_decay` you have tried during your hyperparameter search.

Compare the plots of the 3 different algorithms and briefly explain any differences.

d) Compare the policies you have found with *SARSA*, *Q-Learning*, and *Expected SARSA* with the policy you have found using *Policy Iteration*. Also compare the corresponding value functions v. What do you find? (Make sure to compare plots where γ is same.)

e) In general: Is the optimal policy π^* unique? Is the optimal value function v^* unique? Explain your reasoning.

2 RL with non-tabular Q function [5 Points]

If the state space is not discrete and possibly infinite, it is not possible to use the state × action Q table. It is instead necessary to make use of function approximators such as neural networks. Typically, if the state is a vector in $\mathbb{R}^{n_{state}}$, the simplest choice is to assume that Q is linear: $Q_\theta(s,a) = \theta_a^T s$ where θ is a matrix of parameters of size $n_{state} \times n_{action}$ and each of its rows are associated with an action and are written θ_a^T of size n_{state}. Other choices are possible. For instance, it is beneficial to make assumptions of Q functions that are adapted to the structure of the state vectors. In question c), the Q function will be approximated by a neural network.

The algorithms for on-policy and off-policy TD learning have to change. The main difference is that instead of the update $Q(s,a) \leftarrow Q(s,a) + \alpha(y - Q(s,a))$[2] we perform a gradient descent step to minimize the error of prediction of the return $E = \frac{1}{2}(Q(s,a) - y)^2$.

a) Show that, using a linear model $Q_\theta(s,a) = \theta_a^T s$, the update of the Q table is replaced by the parameters updates:

- if y is assumed constant with respect to θ we have $\theta_a \leftarrow \theta_a - \alpha(Q_\theta(s,a) - y)s$,
- if $y = r + \gamma Q_\theta(s',a')$ it becomes $\begin{cases} \theta_a \leftarrow \theta_a - \alpha(Q_\theta(s,a) - y)s, \\ \theta_{a'} \leftarrow \theta_{a'} + \alpha\gamma(Q_\theta(s,a) - y)s', \end{cases}$

b) Assume you use the update rule

$$\theta_a \leftarrow \theta_a - \alpha(Q_\theta(s,a) - y)s \tag{1}$$

with $y = r + \gamma Q_\theta(s',a')$ and $Q_\theta(s,a) = \theta_a^T s$. Prove or disprove the following statement:

"Assume we are given a grid world environment such as `FrozenLake` with states $\mathcal{S} = \{s_1, \ldots, s_{n_{state}}\}$. If we artificially rewrite the state vector s which corresponds to state s_i as a vector of $\mathbb{R}^{n_{state}}$ with 1 at position i and zero elsewhere (one-hot encoding), the learning rule in Equation 1 is effectively equivalent to SARSA with a standard Q table."

c) We will use PyTorch[3] to solve the MountainCar environment. Familiarize yourself with the environment via the documentation[4]. In the file `ex2_mountain_car_Q_learning.py` implement Q-Learning with a linear model. Read the code carefully and fill in the corresponding TODOs. Starting with $\varepsilon = 1$, pick reasonable hyperparameters for γ, α and `eps_decay`, train your agent for an appropriate number of episodes (e.g., 2000) and test it using 100 episodes. During training, feel free to modify `max_episode_length`. During testing, set `max_episode_length=200` and again pick actions deterministically.

The code produces plots that show the accumulated rewards over training and test episodes, the loss over the training episodes, and the number of steps the agent needs to reach the goal state over all test episodes (−1 means the goal was not reached after `max_episode_length`).

[2] y is the one-step-ahead prediction of the return based on TD, in SARSA it is written $y = r + \gamma Q(s',a')$ with s', a' the next state/action pair, in Q-Learning it is $y = r + \gamma \max_{a'} Q(s',a')$

[3] https://pytorch.org/get-started/

[4] https://gymnasium.farama.org/environments/classic_control/mountain_car/

In this setting, the linear model will perform very poorly on this task. One of the reasons for this is the *sparse reward structure* of this task. Try to change the reward to represent a more informative quantity for the agent (see `custom_reward` method). The aim is to create an agent that can navigate the car to the goal state in a couple of episodes.

If your agent still performs poorly, feel free to modify the code to change the state representation: Right now, the state s is a 2D vector ($s \in \mathbb{R}^2$). Can you think of transformations to the vector s that make it easier for a linear model to perform better on this task?

Report all transformations you apply to both the reward and the state and reason about why you chose them. Include all plots of your final agent in your report and report how you chose the hyperparameters.

d) Fill in the TODOs in `ex2_mountain_car_Q_learning.py` that implement a *non-linear* model Q_θ to solve this task. Specifically, implement a neural network with one hidden layer as described below. If s is the observed state vector of size n_{state}, and n_{hidden} is your number of hidden neurons:

- The activation of the hidden layer is given by $a_y = W_{hid}s + b_{hid}$ where W_{hid} is a variable matrix of size (n_{hidden}, n_{state}) and b_{hid} is a variable vector of size n_{hidden}.
- The output of the hidden layer is defined as $y = \text{torch.relu}(a_y)$ (`relu` computes the rectified linear function, which is 0 if x is negative and x otherwise). Try different activation functions such as `torch.tanh`.
- The activation of the output layer is given by $a_z = wy + b$ where w is a matrix of size (n_{action}, n_{hidden}) and b is a vector of size (n_{action}).

All steps above should be implemented using pre-defined PyTorch modules (`Sequential`, `Linear`, ...) in combination with the Adam optimizer (as in Task b). The goal is to train an agent that reaches the goal state in **all** 100 test episodes. You can still use your custom reward function from Task b. However, if you have implemented a feature transformation of the state s, remove it for this task (i.e., the neural network's input is the state vector $s \in \mathbb{R}^2$ which the environment outputs). Again, find good hyperparameters and include all plots in your report. Report your findings and compare the results of the linear and the non-linear Q model.