Exercise Sheet 1

Due Date: October 28, 2024

Exercises

In this set of exercises, you will implement the Value Iteration algorithm and tabular methods for solving Markov Decision Processes. Your task is to implement the algorithms, evaluate them on benchmark environments, and answer questions about their performance and properties. **You may work alone or in groups of two.** If you work in a pair, include both names and UČOs in your report.

To pass the homework, you need to get **at least 70 points** out of 100 points.

Implementation (max. 36 points)

Complete the incomplete classes in the `hw1.py` file to implement algorithms from the lecture. Submit your code as a single Python file to the file vault (odevzdávárna) called *code_hw1* in the IS.

Environment Wrapper and Logging Note that an environment wrapper (for FrozenLake and CliffWalking) is prepared for you in *infrastructure/envs/tabular_wrapper.py*. The wrapper provides an interface for easily getting dynamics/reward information for both benchmarks. You should read through this interface before starting your implementation. You can also use the `Logger` class to log data during training to a CSV file.

Interface You will complete the corresponding trainer class for each algorithm. A trainer class is required to have the constructor and the `train` method. The **train** method should train a new policy from scratch in a provided environment and return it.

The signatures of prepared methods (positional arguments, **return type**) should remain unchanged for evaluation purposes, however, you are free to add your own methods, or your own training (hyper) parameters via keyword arguments. Just make sure you provide default values for these parameters since we will only supply the positional parameters during the evaluation.

Value Iteration

[**10 points**] Complete the `train` method of the `VITrainer` class in the `hw1.py` file. Use methods `get_dynamics_tensor`, `get_reward_tensor`, `get_transition`, and `get_reward` of the environment wrapper to implement the value iteration algorithm. Initialize the value function $\hat{V}$ with a zero vector.

You can visualize your results via the `render_policy` method of the environment wrapper. Compare the output with the expected result on each map, located in the `images` folder. All results were generated with $\gamma = 0.99$ and steps $= 500$.

[**5 points**] Answer the following questions:

- On which of the provided environments does the algorithm produce the optimal value function in finitely many steps? Why?

- On which of the provided environments does the algorithm produce the optimal policy in finitely many steps? Why?

Tabular Algorithms

[**21 points**] Complete the `train` method of trainer classes `QLTrainer`, `SARSATrainer`, and `MCTrainer` to implement Q-Learning, SARSA, and Every-Visit Monte Carlo Control algorithms, respectively.

Use methods `step` and `reset` of the environment wrapper to interact with the environment. In `QLTrainer`, you can see a sketch of the training loop. Note that you can pass an instance of the `Logger` class to the `train` method to log additional data during training.

For all three algorithms, use the ε-greedy policy for exploration.

Evaluation (max. 64 points)

Evaluate your implementation on environment `FrozenLake`, `LargeLake`, and `CliffWalking` prepared in the `hw1.py` file. Then answer the following question in a report submitted to the file vault (odevzdávárna) called *report_hw1* in the IS.

- Train each algorithm on 100000 samples from the environment. Use the following hyperparameters: $\gamma = 0.99$, $\alpha = 0.1$, and $\varepsilon = 0.1$.
- Repeat the training process multiple times.
- [**16 points**] Plot how the average reward accumulated per episode and value estimate $\hat{Q}(s_0)$ of the initial state change over time. Use whatever plot you think is the most informative. However, the plot should ideally display more than merely average values [**+6 points**].

Compare the performance of the algorithms and answer the following questions:

- [**5 points**] How do you decide what number of training repetitions is sufficient to evaluate the performance of an algorithm?
- [**5 points**] Do the algorithms converge to the optimal value function? If not, why? What would happen if we decreased ε over time? Would some of the algorithms behave differently?
- [**5 points**] Explain the reasons for differences in the speed of convergence between the algorithms.
- [**5 points**] What advantages and drawbacks of each algorithm do you observe?

Optimize hyperparameters:

- [**11 points**] Try different values of ε and α. What are your findings?

Evaluate algorithms with exploring starts:

- Use method `env.reset(randomize=True)` to start each episode in a random state.
- [**11 points**] How does the performance of the algorithms change with exploring starts? Why?

Algorithms Overview

Value Iteration (Lecture 2)

Value Iteration is a dynamic programming algorithm used to find the optimal policy for a Markov Decision Process (MDP). The core idea is to iteratively update the estimate $\hat{V}$ of the optimal value function $V(s)$ for each state s using the Bellman equation:

$$\hat{V}_{k+1}(s) = \max_a \sum_{s'} p(s' \mid s, a) \left[r(s, a, s') + \gamma \hat{V}_k(s') \right],$$

where $p(s' \mid s, a)$ is the transition probability, $r(s, a, s')$ is the reward, and γ is the discount factor. By the policy improvement theorem, the greedy policy recovered from the value estimates

$$\pi(s) = \arg\max_a \sum_{s'} p(s' \mid s, a) \left[r(s, a, s') + \gamma \hat{V}(s') \right]$$

is guaranteed to achieve at least $\hat{V}(s)$ for all states s.

Every-Visit Monte Carlo Control (Lecture 3)

Every-Visit Monte Carlo Control is a model-free algorithm that estimates the action-value function $Q(s, a)$ by averaging the returns observed after visiting state-action pairs in episodes. For each episode, the return G_t is calculated as:

$$G_t = \sum_{k=t}^{T-1} \gamma^{k-t} r_{k+1},$$

where T is the final timestep of the episode. For every timestep t, the action-value function is updated as:

$$N(s_t, a_t) \leftarrow N(s_t, a_t) + 1$$
$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \frac{1}{N(s_t, a_t)} \left[G_t - \hat{Q}(s_t, a_t) \right]$$

To ensure exploration, an ε-greedy policy is typically used to select actions during the episode.

Q-Learning (Lecture 4)

Q-learning is a model-free, off-policy reinforcement learning algorithm that learns the optimal action-value function $Q(s, a)$. It maintains an estimate $\hat{Q}(s, a)$ of the true action-value function, which is iteratively refined. In Q-Learning, the agent interacts with the environment using a behavioural policy, typically an ε-greedy policy (with respect to $\hat{Q}$), which balances exploration and exploitation. The agent collects samples in the form of (s, a, r, s'), where s is the current state, a is the action taken, r is the reward received, and s' is the next state. The update rule for the action-value function estimate $\hat{Q}(s, a)$ is:

$$\hat{Q}(s, a) \leftarrow \hat{Q}(s, a) + \alpha \left[r + \gamma \max_{a'} \hat{Q}(s', a') - \hat{Q}(s, a) \right],$$

where α is the learning rate, and γ is the discount factor. The key feature of Q-Learning is that it updates $\hat{Q}(s, a)$ using the maximum future reward from the next state, allowing it to learn the optimal policy $\pi^*(s)$, even though the samples are generated from a different (behavioural) policy.

SARSA (Lecture 4)

SARSA is a model-free, on-policy reinforcement learning algorithm that updates the action-value function $\hat{Q}(s, a)$ using the agent's own experience with the environment. The agent follows a behavioural policy, typically ε-greedy, to interact with the environment and generates samples in the form of (s, a, r, s', a'), where s is the current state, a is the action taken, r is the reward received, s' is the next state, and a' is the action taken in the next state. The update rule for the action-value function $\hat{Q}(s, a)$ is:

$$\hat{Q}(s, a) \leftarrow \hat{Q}(s, a) + \alpha \left[r + \gamma \hat{Q}(s', a') - \hat{Q}(s, a) \right]$$

In contrast to Q-Learning, SARSA updates $\hat{Q}(s, a)$ based on the action a' actually taken in the next state s'.