Exercise 5
Model-free Control

Two weeks ago you implemented a simple gridworld MDP that adheres to the OpenAI Gym interface. This week we will put your implementation into practice and try to solve it using SARSA and Q-Learning.

1 SARSA and Q-Learning

You can find the entry point inside the `td_control.py` file. We provide to you a sample implementation of the environment from Exercise 3 in `gridworld.py`. Visualization and testing helpers can be found inside `utils.py`. The actual TD-control routines can be found in the `agents` folder.

- `base_agent.BaseAgent`: The base class, which has the `__init__` constructor and an `action` function, which returns the ϵ-greedy action for a state s.
- `sarsa.SARSAAgent` and `qlearning.QLearningAgent`: The SARSA and Q-learning agent classes with methods `learn` and `update_Q`. Both inherit from `BaseAgent`.

Remark: You are free to implement the inner workings of your agents as you wish. For the visualization tools to work you will however have to work with the provided Q-value member variable Q, which is a numpy array of shape $[grid\_height, grid\_width, num\_actions]$. Alternatively, it should be easy to adjust the visualization helper functions as needed.

Programming Tasks:

1. **ϵ-greedy actions**: Implement the `action` function inside `BaseAgent`, which should return a random action with a probability of ϵ, and the greedy action w.r.t the current Q-value estimates of state s with a probability of $1 - \epsilon$.
2. **SARSA**, implementation to be done in `SARSAAgent`, can be run with the flag `-agent=sarsa`:
   - **Q-value update**: Implement the Q-value update rule of SARSA for a tuple (s, a, r, s', a') inside `update_Q`.
   - **Learning loop**: Implement the training loop of the SARSA agent, which should interact with the environment for `n_timesteps` steps.
3. **Q-learning**, implementation to be done in `QLearningAgent`, can be run with the flag `-agent=qlearning`: `SARSAAgent`, can be run with the flag `-agent=sarsa`:
   - **Q-value update**: Implement the Q-value update rule of Q-learning for a tuple (s, a, r, s') inside `update_Q`.
   - **Learning loop**: Implement the training loop of the Q-learning agent, which should interact with the environment for `n_timesteps` steps.

Test your implementation and verify that it's working correctly. Feel free to play around with the hyperparameters (for details see `td_control.parse` function). You can see a sample result in Fig. 1.

2 CliffWalking

In order to highlight the difference between SARSA and Q-Learning, try to replicate the famous `CliffWalking` environment by extending the gridworld implementation (Fig. 2). For simplicity, we replicated this environment inside a 4 × 4 grid with only two cliff cells between start and finish.

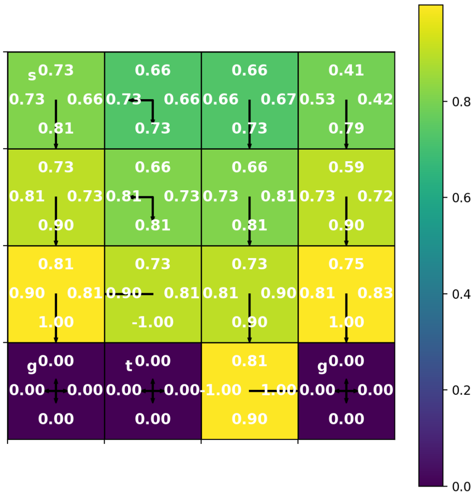

Figure 1: Sample output for a Q-Learning agent on the gridworld environment ($200k$ training steps, $\epsilon = 0.4$, $\gamma = 0.9$, $\alpha = 0.05$). States $(1, 2)$ and $(2, 2)$ should actually have a $50 - 50$ policy for actions down and right, which they don't have due to rounding error. Results can also be quite sensitive to hyper-parameters and a correct implementation can still lead to confusing results sometimes. Especially the Q-values for states far away from the terminal states vary over runs, due to involved stochasticity.

Programming Tasks:

1. **CliffWalk**: Implement a 4×4 version of the `CliffWalk` environment inside `gridworld.py`. Make sure, that a small negative reward is returned for each step (and test what happens without).

2. **Train** a SARSA and Q-Learning agent on this environment. The environment can be run with the flag `-env=cliffwalk`
   - What is the explanation for the difference in learnt policies and Q-functions?
   - How does this relate to SARSA being considered an on-policy method and Q-Learning being an off-policy method?

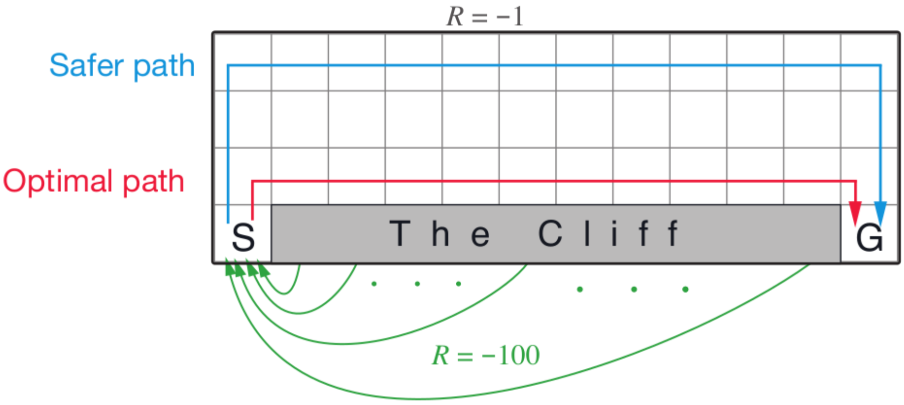

Figure 2: The `CliffWalking` environment. We will replicate a 4×4 version of this setup in our implementation. Each non-terminal state returns a small negative reward to encourage short paths.