COMP3667: Reinforcement learning practical 4

Dynamic Programming

robert.lieck@durham.ac.uk

1 Overview

Welcome to the fourth reinforcement learning practical. In this practical, we will dive a bit more into the practical details of dynamic programming methods. In particular, we will:

- Use the "Frozen Lake" environment as a basic example.
- Implement *policy evaluation* and understand how this algorithm updates state values.
- Implement *policy improvement* to learn better policies.
- Implement *policy iteration* to learn optimal policies.

For this practical, you will need a basic Python environment with `numpy` and `gymnasium`.

2 Frozen Lake Environment

You can install the required dependencies via the `rldurham` package if you have not done this already.

```
!pip install swig
!pip install --upgrade rldurham
```

Do the relevant imports

```python
import gymnasium as gym  # you don't need this: use rld.make instead of gym.make
import numpy as np
import rldurham as rld
```

and load the Frozen Lake environment

```python
env = rld.make(
    'FrozenLake-v1',          # small version
    # 'FrozenLake8x8-v1',     # larger version
    # desc=["GFFS", "FHFH", "FFFH", "HFFG"],  # custom map
    render_mode="rgb_array", # for rendering as image/video
    is_slippery=False,       # warning: slippery=True results in complex dynamics
)
rld.env_info(env, print_out=True)
rld.seed_everything(42, env)
LEFT, DOWN, RIGHT, UP = 0, 1, 2, 3
```

```
Seed set to 42
actions are discrete with 4 dimensions/#actions (action space: Discrete(4))
observations are discrete with 16 dimensions/#observations (observation space: Discrete(16))
maximum timesteps is: 100
```

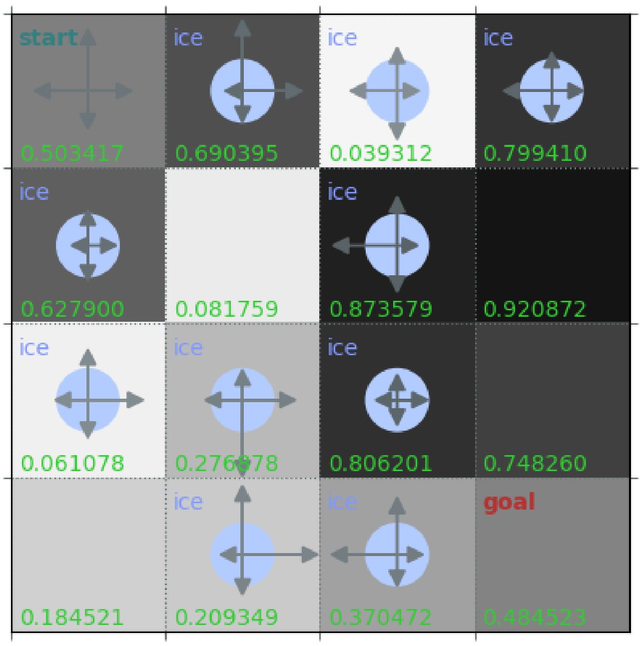
The documentation for the environment is at `https://gymnasium.farama.org/environments/toy_text/frozen_lake/`. Use `slippery=False` for now, because otherwise the optimal solutions are not very intuitive (with `slippery=True`, when the agent tries to move forward it will with equal probability instead slip sideways, but never backwards; therefore it is sometimes better to try to move "away from a hole" instead of "towards the goal").

You can render the environment's state using its internal visualisation (requires `render_mode="rgb_array"`)

```
rld.render(env)
```

or use this helper function, which can also show policies and value functions

```
rld.plot_frozenlake(env=env,
                    v=np.random.uniform(0, 1, 16),
                    policy=np.random.uniform(0, 1, (16, 4)),
                    draw_vals=True)
```



- Define a uniform policy and plot it in the environment. *Hint:* `env.observation_space.n` and `env.action_space.n` gives you the number of states and actions, respectively; you can provide the policy to the plotting function via the `policy` argument.

3 Policy Evaluation

Now, we would like to know how well a policy performs, that is, what the state value of a particular state is when following the policy in the future. This *policy evaluation* procedure can be efficiently done using dynamic programming, which iteratively improves state value estimates and converges to the true state values.

- Write a `policy_eval_step` function that takes an initial state value estimate $v_\pi^k(s)$ and computes an improved estimate $v_\pi^{k+1}(s)$ as

$$v_\pi^{k+1}(s) = \sum_a \pi(a \mid s) \sum_{s'} p(s' \mid s, a) \left[\mathcal{R}_{ss'}^a + \gamma\, v_\pi^k(s')\right] \ . \tag{1}$$

  Hint: `env.P[s][a]`: is giving you a list of tuples `(p, s', r, done)`, one for each possible transition from state `s` when taking action `a`: `p` is the probability of this transition to happen, `s'` is the state the agent transitions to, `r` is the reward it receives, and `done` indicates whether the episode is over (which you will not need to use).

- Initialise the state values with zero, take one `policy_eval_step` at a time and plot the result to observe how updates are being performed. *Hint:* You can provide the state values to the plot function using the `v` argument, specifying `draw_vals=True` additionally shows the numeric state values.

- On an intuitive level, how would you describe the dynamics you see? What seems to be inefficient about the current implementation? How could this be improved?

- Implement a modified `policy_eval_step_inplace` version of the function that performs state value updates in-place. That is, instead of clearly separating the "old" values $v_\pi^k(s)$ and the "new" values $v_\pi^{k+1}(s)$, you operate on a single state value estimate $v_\pi(s)$, which is updated as you go. *Hint:* Make sure the updates for a particular state are still "atomic" and you do not use the half-computed values (in case of transitions that *stay* in a particular state).
  - Think about a clever order in which to update state values in-place. *Hint:* States are ordered from top-left (start: 0) to bottom-right (goal: 15).
  - Again, observe the step-wise update of values from one iteration to the next. How does that compare to the original implementation?

- Write a `policy_evaluation` function that iteratively updates the state values using the `policy_eval_step` or `policy_eval_step_inplace` function and stops if they do not change (by some small tolerance value). Print the number of iterations needed to converge and compare for the `policy_eval_step` and `policy_eval_step_inplace` implementation.

- How many iterations do you need until state values have converged to their true value? To make it simpler, do the following though experiment: Take an environment that has only a single state and a single action (so nothing can really change and there is only one possible policy) and you get a reward of 1 upon every transition. Take the update equation for the state value from above, which now simplifies to

$$v^{k+1} = 1 + \gamma\, v^k \ . \tag{2}$$

  Can you write down v^n in a non-recursive form assuming you start with $v^0 = 0$? Can you write down v^∞ in closed form? Is v^∞ the exact state value? How long does it take to converge? What happens for $\gamma = 1$ as opposed to $\gamma < 1$?

4 Policy Improvement

- Implement a function that computes state-action values $q_\pi(s, a)$ (for all actions in a given state) from the state values $v_\pi(s)$ using their known relation

$$q_\pi(s, a) = \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \left[\mathcal{R}_{ss'}^a + \gamma\, v_\pi(s')\right] \ . \tag{3}$$

- Implement a `policy_improvement` function that takes a state value estimate and defines a policy (by first computing state-action values) that achieves maximal return (i.e. always chooses an action with maximal state-action value). Optionally, either choose actions deterministically, or choose all actions with maximum value with the same probability (if there is more than one such action).

- Load the larger `'FrozenLake8x8-v1'` environment (again with `is_slippery=False` for now), compute state values by evaluating the uniform policy, plot the result. Then compute an improved policy and plot the result again.

5 Policy Iteration

- Starting from the improved policy from above, perform two updates by doing
  - policy evaluation (plot the results)
  - policy improvement (plot the results).

  Use a stochastic policy improvement (i.e. choose optimal action with equal probability) and a discount value of $\gamma = 1$. What do you observe? To you see anything that could be problematic? Can you explain what you observe (you may need to print the raw state values)? Would a deterministic policy improvement help? *Hint:* Remember what we learned above about value convergence and paths of finite/infinite length.

- Change the value to $\gamma = 0.999$ and do another two updates. What is different? *Hint:* You can use $\gamma = 0.9$ to see the effect more clearly.
