COMP3667: Reinforcement learning practical 4

Dynamic Programming

robert.lieck@durham.ac.uk & christopher.g.willcocks@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 OpenAI Gym "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`, `matplotlib`, and OpenAI `gym` (version 0.20.0).

2 Frozen Lake Environment

Install OpenAI `gym` if you have not done this already. You will need to use version 0.20.0: </> copy code

```
%%capture
!pip install setuptools==65.5.0 "wheel<0.40.0"
!apt update
!pip install 'gym==0.20.0'
```

Do the relevant imports </> copy code

```
import gym
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.font_manager
```

and load the Frozen Lake environment </> copy code

```
name = 'FrozenLake-v1'  # small version
# name = 'FrozenLake8x8-v1'  # larger version
env = gym.make(name, is_slippery=False)
env.seed(742)
env.action_space.seed(742)
# named actions
LEFT, DOWN, RIGHT, UP = 0, 1, 2, 3
```

The code of the environment is at `https://github.com/openai/gym/blob/master/gym/envs/toy_text/frozen_lake.py`. 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"). Use this helper function to visualise the environment in a nice way: </> copy code

```
def plot(env, v=None, policy=None, col_ramp=1, dpi=175, draw_vals=False, mark_ice=True):
    # set up plot
    plt.rcParams['figure.dpi'] = dpi
    plt.rcParams.update({'axes.edgecolor': (0.32,0.36,0.38)})
```

```
    plt.rcParams.update({'font.size': 4 if env.env.nrow == 8 else 7})
    gray = np.array((0.32,0.36,0.38))
    plt.figure(figsize=(3, 3))
    ax = plt.gca()
    ax.set_xticks(np.arange(env.env.ncol)-.5)
    ax.set_yticks(np.arange(env.env.nrow)-.5)
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    plt.grid(color=(0.42,0.46,0.48), linestyle=':')
    ax.set_axisbelow(True)
    ax.tick_params(color=(0.42,0.46,0.48),
                  which='both', top='off', left='off', right='off', bottom='off')
    # use zero value as dummy if not provided
    if v is None:
        v = np.zeros(env.nS)
    # plot values
    plt.imshow(1-v.reshape(env.env.nrow,env.env.ncol)**col_ramp,
              cmap='gray', interpolation='none',
              clim=(0,1), zorder=-1)
    # go through states
    for s in range(env.nS):
        x, y = s % env.env.nrow, s // env.env.ncol
        # print numeric values
        if draw_vals and v[s] > 0:
            vstr = '{0:.1e}'.format(v[s]) if env.env.nrow == 8 else '{0:.6f}'.format(v[s])
            plt.text(x - 0.45, y + 0.45, vstr, color=(0.2, 0.8, 0.2), fontname='Sans')
        # mark ice, start, goal
        if env.desc.tolist()[y][x] == b'F':
            plt.text(x-0.45,y-0.3, 'ice', color=(0.5, 0.6, 1), fontname='Sans')
            if mark_ice:
                ax.add_patch(plt.Circle((x, y), 0.2, color=(0.7, 0.8, 1), zorder=0))
        elif env.desc.tolist()[y][x] == b'S':
            plt.text(x-0.45,y-0.3, 'start',color=(0.2,0.5,0.5), fontname='Sans',
                     weight='bold')
        elif env.desc.tolist()[y][x] == b'G':
            plt.text(x-0.45,y-0.3, 'goal', color=(0.7,0.2,0.2), fontname='Sans',
                     weight='bold')
            continue # don't plot policy for goal state
        else:
            continue # don't plot policy for holes
        # plot policy
        def plot_arrow(x, y, dx, dy, v, scale=0.4):
            plt.arrow(x, y, scale * float(dx), scale * float(dy), color=gray+0.2*(1-v),
                      head_width=0.1, head_length=0.1, zorder=1)
        if policy is not None:
            a = policy[s]
            if a[0] > 0.0: plot_arrow(x, y, -a[0],    0., v[s]) # left
            if a[1] > 0.0: plot_arrow(x, y,    0.,  a[1], v[s]) # down
            if a[2] > 0.0: plot_arrow(x, y,  a[2],    0., v[s]) # right
            if a[3] > 0.0: plot_arrow(x, y,    0., -a[3], v[s]) # up
    plt.show()
```

Have a look

</> copy code

```
print('action space: ' + str(env.action_space))
print('reward range: ' + str(env.reward_range))
print('observation space: ' + str(env.observation_space))
plot(env=env)
--> action space: Discrete(4)
--> reward range: (0, 1)
--> observation space: Discrete(16)
```

- Define a uniform policy and plot it in the environment. *Hint:* `env.nS` and `env.nA` give 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.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.
