COMP3667: Reinforcement learning practical 6

Temporal-Difference Learning

robert.lieck@durham.ac.uk

This is the version *with* answers!

Notebook with code

1 Overview

Welcome to the sixth reinforcement learning practical. In this practical, we will be experimenting with different TD methods to better understand their characteristics, advantages, and drawbacks. In particular, we will

- evaluate on-policy SARSA(0) and off-policy Q-Learning
- experiment with n-step TD learning
- compare the performance of these methods in different environments
- understand when/how/why they may fail to work and how to fix them.

2 Setup

```python
import numpy as np
import matplotlib.pyplot as plt
import itertools
from IPython import display
import rldurham as rld
from rldurham import plot_frozenlake as plot
```

As before, we will use different versions of the frozen lake gym environment:

```python
name = 'FrozenLake-v1'
env = rld.make(name, is_slippery=False)                 # 4x4
env = rld.make(name, map_name="8x8", is_slippery=False) # 8x8
# env = rld.make(name, desc=["SFHH",
#                            "HFFH",
#                            "HHFF",
#                            "HHHG",], is_slippery=False) # custom
rld.seed_everything(42, env)
# LEFT, DOWN, RIGHT, UP = 0, 1, 2, 3
```

You can use these two helper classes to define hard-coded policies or policies using Q-values

```python
class QPolicy:
    def __init__(self, Q, epsilon, values=False):
        self.Q = Q
        self.epsilon = epsilon
        self.values = values
    def sample(self, state):
        if np.random.rand() > self.epsilon:
            best_actions = np.argwhere(self.Q[state]==np.max(self.Q[state])).flatten()
```

```python
            return np.random.choice(best_actions)
        else:
            return env.action_space.sample()
    def __getitem__(self, item):
        state, action = item
        if self.values:
            return self.Q[state, action] / (self.Q[state].sum() + 1e-10)
        else:
            best_actions = np.argwhere(self.Q[state]==np.max(self.Q[state])).flatten()
            p = int(action in best_actions) / len(best_actions)
            return (1 - self.epsilon) * p + self.epsilon / len(self.Q[state])

class HardCodedPolicy:
    def __init__(self, state_action_map):
        self.state_action_map = state_action_map
    def sample(self, state):
        if state in self.state_action_map:
            return np.random.choice(self.state_action_map[state])
        else:
            return np.random.choice(4)
    def __getitem__(self, item):
        state, action = item
        if state in self.state_action_map:
            if action in self.state_action_map[state]:
                return 1 / len(self.state_action_map[state])
            else:
                return 0
        else:
            return 1 / 4
```

We can keep some plotting data in these variables (re-evaluate the cell to clear data)

```python
reward_list = [[]]
auc = [0]
test_reward_list = [[]]
test_auc = [0]
plot_data = [[]]
plot_labels = []
experiment_id = 0
```

and use these functions to update and plot the learning progress

```python
# (using global variables in functions)
def update_plot(mod):
    reward_list[experiment_id].append(reward_sum)
    auc[experiment_id] += reward_sum
    test_reward_list[experiment_id].append(test_reward_sum)
    test_auc[experiment_id] += test_reward_sum
    if episode % mod == 0:
        plot_data[experiment_id].append([episode,
                                         np.array(reward_list[experiment_id]).mean(),
                                         np.array(test_reward_list[experiment_id]).mean()])
        reward_list[experiment_id] = []
        test_reward_list[experiment_id] = []
        for i in range(len(plot_data)):
            lines = plt.plot([x[0] for x in plot_data[i]],
                             [x[1] for x in plot_data[i]], '-',
                             label=f"{plot_labels[i]}, AUC: {auc[i]}|{test_auc[i]}")
            color = lines[0].get_color()
            plt.plot([x[0] for x in plot_data[i]],
                     [x[2] for x in plot_data[i]], '--', color=color)
```

```python
        plt.xlabel('Episode number')
        plt.ylabel('Episode reward')
        plt.legend()
        display.clear_output(wait=True)
        plt.show()

def next_experiment():
    reward_list.append([])
    auc.append(0)
    test_reward_list.append([])
    test_auc.append(0)
    plot_data.append([])
    return experiment_id + 1
```

3 On-policy and off-policy learning with TD(0)

Recap TD Learning

Remember our 0-step temporal difference (TD) targets from the lecture, which can be computed for any (also partial) episodes

$$
\begin{aligned}
V_\pi(s_t) &= \sum_{a_t \in \mathcal{A}} \pi(a_t \,|\, s_t) \sum_{s_{t+1} \in \mathcal{S}} p(s_{t+1} \,|\, s_t, a_t) \Big[\mathcal{R}^{a_t}_{s_t s_{t+1}} + \gamma \, V_\pi(s_{t+1}) \Big] \\
&= \mathbb{E}_{a_t \sim \pi(a_t \,|\, s_t)} \mathbb{E}_{s_{t+1} \sim p(s_{t+1} \,|\, s_t, a_t)} \Big[\mathcal{R}^{a_t}_{s_t s_{t+1}} + \gamma \, V_\pi(s_{t+1}) \Big] \\
&\approx \underbrace{\mathcal{R}^{a_t}_{s_t s_{t+1}} + \gamma \, V_\pi(s_{t+1})}_{\text{TD(0) target}} \\
Q_{\pi^*}(s_t, a_t) &= \sum_{s_{t+1} \in \mathcal{S}} p(s_{t+1} \,|\, s_t, a_t) \Big[\mathcal{R}^{a_t}_{s_t s_{t+1}} + \gamma \sum_{a_{t+1} \in \mathcal{A}} \pi^*(a_{t+1} \,|\, s_{t+1}) \, Q_{\pi^*}(s_{t+1}, a_{t+1}) \Big] \\
&= \mathbb{E}_{s_{t+1} \sim p(s_{t+1} \,|\, s_t, a_t)} \Big[\mathcal{R}^{a_t}_{s_t s_{t+1}} + \gamma \, \mathbb{E}_{a_{t+1} \sim \pi^*(a_{t+1} \,|\, s_{t+1})} Q_{\pi^*}(s_{t+1}, a_{t+1}) \Big] \\
&\approx \underbrace{\mathcal{R}^{a_t}_{s_t s_{t+1}} + \gamma \, Q_{\pi^*}(s_{t+1}, a_{t+1})}_{\text{TD(0) target}} \; .
\end{aligned}
$$

At a particular time t we are in state s_t and take action $a_t \sim \pi(a_t \,|\, s_t)$ sampled from the policy π. We then end up in state $s_{t+1} \sim p(s_{t+1} \,|\, s_t, a_t)$ based on the environment's transition function. If we are interested in learning state-action values (i.e. solving the control problem), we additionally need to consider the following action $a_{t+1} \sim \pi^*(a_{t+1} \,|\, s_{t+1})$ based on the policy π^* that we want to learn or evaluate. Note that the sampling policy π and the policy $\pi*$ we want to evaluate are the same in *on-policy* methods but may be different in *off-policy* methods.

The TD(0) targets are noisy "snapshots" of how the values should look like based on the current transition at time t. The difference between the TD target and our current value estimate gives us a noisy TD error signal that tells us "how far off" our estimates are. This can be use to update our value estimates with a learning rate α (similar to SGD) to improve them

$$
\begin{aligned}
V_\pi(s_t) &\leftarrow V_\pi(s_t) + \alpha \big[\underbrace{\mathcal{R}^{a_t}_{s_t s_{t+1}} + \gamma \, V_\pi(s_{t+1})}_{\text{TD(0) target}} - V_\pi(s_t) \big] \\
Q_{\pi^*}(s_t, a_t) &\leftarrow Q_{\pi^*}(s_t, a_t) + \alpha \big[\underbrace{\mathcal{R}^{a_t}_{s_t s_{t+1}} + \gamma \, Q_{\pi^*}(s_{t+1}, a_{t+1})}_{\text{TD(0) target}} - Q_{\pi^*}(s_t, a_t) \big] \; .
\end{aligned}
$$

On-policy SARSA(0)

We can evaluate and improve our policy "on the go". This is called *on-policy* learning and it means that the policy π we use for sampling is the same as the policy π^* we are evaluating and learning.

Exercise

- Implement SARSA(0) by filling in the TD(0) targets and updates in the skeleton below. (The rest of the skeleton is for collecting episodes, evaluating the learned policy and plotting everything. The dashed line is the reward for the learned policy, the solid line is for the sampling policy.)
- Run a couple of evaluations with different values for the learning rate `alpha` and the exploration `epsilon` on the 4x4 environment. Use both "noisy" and "neutral" initialisations for Q (commenting in/out the respective lines in the code). What effects do you observe? With what parameters does the agent learn best?

  In deterministic environments with a single goal state (like FrozenLake), learning typically occurs as a "jump". Once the agent improves, it jumps directly to the level of its final performance (this is because there is only one simple goal state; either it knows how to get there or not). "Good" learning performance means that the agent makes that jump early (with a high probability). The best performance is achieved with intermediate values for both learning rate and exploration.

  Learning rate alpha: Very large (approaching 1) and very low (below 0.01) learning rates are detrimental. Large learning rates make the convergence unstable; small learning rates are not a problem with a "neutral" initialisation but delay learning with a "noisy" initialisation.

  Exploration epsilon: Generally, higher exploration results in a lower final performance because the agent does not act optimally. With "neutral" initialisation, little exploration is not detrimental (the policy chooses random actions if all values are zero; if they are not zero anymore, the agent has found the solution and we do not need to explore; this is again due to having only a single goal state). With a "noisy" initialisation, little exploration delays or prevents learning. Little exploration may also result in some states of the environment not being explored (if they are not on the found "solution path") and thus not having a good policy for these states.

$\epsilon = 0.1$

$\epsilon = 0.5$

```python
# parameters
num_episodes = 3000
alpha = 0.1
gamma = 0.9
epsilon = 0.5

Q = np.random.uniform(0, 1e-5, [env.observation_space.n, env.action_space.n]) # noisy
Q = np.zeros([env.observation_space.n, env.action_space.n])                    # neutral
V = np.zeros([env.observation_space.n])

# policies
sample_policy = QPolicy(Q, epsilon)
learned_policy = sample_policy
plot_labels.append(f"SARSA (alpha={alpha}, epsilon={epsilon})")

for episode in range(num_episodes):
    state = env.reset()
    reward_sum = 0
```

```
    # learning a policy
    for t in itertools.count():
        action = sample_policy.sample(state)
        next_state, reward, done, _ = env.step(action)
        next_action = learned_policy.sample(next_state)
        # TD(0) targets
        v_target = ...                  # FILL IN HERE!
        q_target = ...                  # FILL IN HERE!
        # updates
        s, a = state, action
        V[s] += ...                     # FILL IN HERE!
        Q[s, a] += ...                  # FILL IN HERE!

        reward_sum += reward
        if done:
            break
        state = next_state

    # testing the learned policy
    state = env.reset()
    test_reward_sum = 0
    while True:
        action = learned_policy.sample(state)
        next_state, reward, done, _ = env.step(action)
        test_reward_sum += reward
        state = next_state
        if done:
            break

    update_plot(int(np.ceil(num_episodes / 20)))

env.close()
experiment_id = next_experiment()
print("Sampling policy and values")
plot(env, v=V, policy=sample_policy, draw_vals=True)
print("Learned policy and optimal/max values")
plot(env, v=Q.max(axis=1), policy=learned_policy, draw_vals=True)
```

Answer: copy code

```
# TD(0) targets
v_target = reward + gamma * V[next_state]
q_target = reward + gamma * Q[next_state, next_action]
# updates
s, a = state, action
V[s] += alpha * (v_target - V[s])
Q[s, a] += alpha * (q_target - Q[s, a])
```

Off-policy Q-Learning

Sometimes, we would like to generate samples with one policy π (typically an exploratory policy) but then use the samples to learn another policy π^* (typically a near-optimal policy). This is possible with *off-policy* methods, such as Q-Learning.

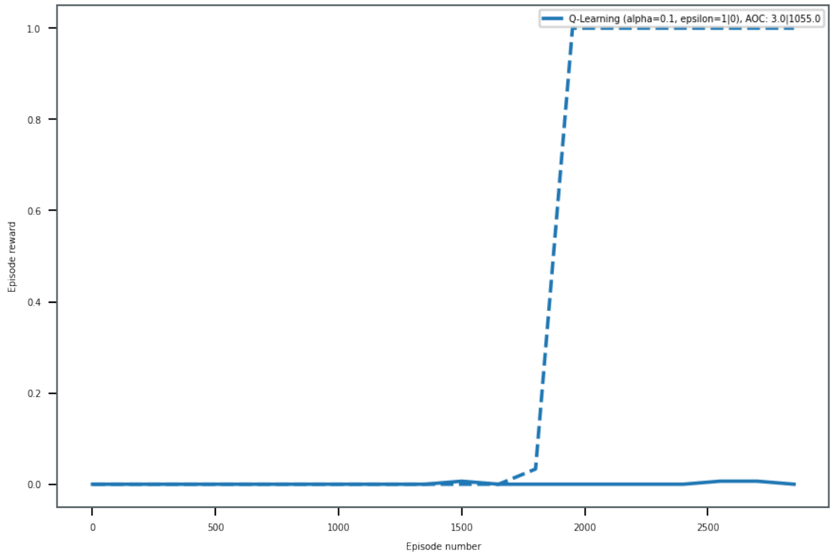
3.0.1 Exercise

- Test Q-Learning by using a different policy for the `learned_policy` (with lower `epsilon`) then for the `sample_policy`.
  - *Note:* Strictly speaking, the `sample_policy` does not change in Q-Learning. You can achieve this by using `epsilon=1` in the `sample_policy` to select actions randomly. For other values of `epsilon`, the

  `sample_policy` "peeks" into the values of the learned policy, which lets it profit from that learning (but is not a clean implementation).
  - *Bonus exercise:* Learn the `sample_policy` using SARSA(0) while learning the `learned_policy` with Q-Learning (this requires maintaining two copies of values estimates, performing separate updates for both etc).

- Run the evaluation several times on the 4x4 and 8x8 environment. Try to get the fastest and most reliable learning by tweaking `epsilon`. How does the performance (episode reward) of the sampling policy (solid line) compare to that of the learned policy (dashed lines)? How is this different from SARSA(0)?

  In SARSA(0) both policies performed equally well (up to noise). Here, the learned policy performs much better than the sampling policy. Even with a completely exploratory sampling policy, the learned policy performs optimally.



Expected TD(0) targets

The normal TD targets to estimate state-action values are computed by *sampling* from the policy π^* that is to be learned. This sampling step increases the noise in the TD error signal (in addition to the noise we already have due to sampling episodes). Instead, one can use the *expected* TD targets

$$\underbrace{\mathcal{R}^{a_t}_{s_t s_{t+1}} + \gamma \sum_{a_{t+1} \in \mathcal{A}} \pi^*(a_{t+1} \,|\, s_{t+1}) \, Q_{\pi^*}(s_{t+1}, a_{t+1})}_{\text{expected TD(0) target}} \ .$$

Exercise

- In the `q_target` replace the sampled value with an expectation over possible actions the `learned_policy` could take. *Note:* `learned_policy[next_state]` gives you the action probabilities for `next_state`.

  Replace

```
q_target = reward + gamma * Q[next_state, next_action]
```

  with

```
expected_Q = (learned_policy[next_state] * Q[next_state]).sum()
q_target = reward + gamma * expected_Q
```

- Compare the performance when using the expected TD(0) target to using the sampled one.

  There should be a slight performance gain, that is, on average the agent jumps to the optimal policy earlier. However, this could be difficult to spot due to the noise of when exactly that jump happens.

4 TD(n)

In TD(0) we do not look ahead and receiving a reward does only affect the value estimate of the current state and action. All the future expected rewards are approximated by using the current value estimates. However, we can improve on that by instead look n steps ahead (in hindsight) and taking the n next steps into account for computing TD targets. These TD(n) targets are

$$V_\pi(s_t) \approx \underbrace{\overbrace{\mathcal{R}^{a_t}_{s_t s_{t+1}} + \gamma\, \mathcal{R}^{a_{t+1}}_{s_{t+1} s_{t+2}} + \ldots + \gamma^n\, \mathcal{R}^{a_{t+n}}_{s_{t+n} s_{t+n+1}}}^{n\text{-step return}} + \gamma^{n+1}\, V_\pi(s_{t+n+1})}_{\text{TD}(n)\text{ target}}$$

$$Q_{\pi^*}(s_t, a_t) \approx \underbrace{\mathcal{R}^{a_t}_{s_t s_{t+1}} + \gamma\, \mathcal{R}^{a_{t+1}}_{s_{t+1} s_{t+2}} + \ldots + \gamma^n\, \mathcal{R}^{a_{t+n}}_{s_{t+n} s_{t+n+1}} + \gamma^{n+1}\, Q_{\pi^*}(s_{t+n+1}, a_{t+n+1})}_{\text{TD}(n)\text{ target}} \;.$$

To compute them in hindsight, we have to store a trace of the last $n+1$ transitions (including the current transition that is also used in TD(0)). The modified skeleton below is providing a trace of length $n+1$ containing the last n steps and the current transition (for $n = 0$ this reduces to the TD(0) case of only considering the current transition).

```python
# parameters
num_episodes = 3000
alpha = 0.1
gamma = 0.9
epsilon = 0.5
on_policy = True   # SARSA or Q-Learning
n = 2              # length of trace to use

# value initialisation
Q = np.random.uniform(0, 1e-5, [env.observation_space.n, env.action_space.n]) # noisy
Q = np.zeros([env.observation_space.n, env.action_space.n])                    # neutral
V = np.zeros([env.observation_space.n])

if on_policy:
    # policies for SARSA
    # vvvvvvvvvvvvvvvvvv
    sample_policy = QPolicy(Q, epsilon)
    learned_policy = sample_policy
    plot_labels.append(f"SARSA (n={n}, alpha={alpha}, epsilon={epsilon})")
    # ^^^^^^^^^^^^^^^^^^
else:
    # policies for Q-Learning
    # vvvvvvvvvvvvvvvvvvvvvvvv
    sample_policy = QPolicy(Q, epsilon)
    td_epsilon = 0.01
    learned_policy = QPolicy(Q, td_epsilon)
    plot_labels.append(f"Q-Learning (n={n}, alpha={alpha}, epsilon={epsilon}|{td_epsilon})")
    # ^^^^^^^^^^^^^^^^^^^^^^^^

for episode in range(num_episodes):
    state, _ = env.reset()
    reward_sum = 0
    done_n = 0

    # trace of the last n + 1 transitions (state, action, reward, next_action)
    trace = np.zeros((n + 1, 4), dtype=int)

    # learning a policy
    for t in itertools.count():
        action = sample_policy.sample(state)
        next_state, reward, term, trun, _ = env.step(action)
        done = term or trun
```

```
        next_action = learned_policy.sample(next_state)

        # remember transitions (incl. next action sampled by learned policy)
        trace[-1] = (state, action, reward, next_action)

        # start computing updates if trace is long enough
        if t > n:

            # n-step targets
            # vvvvvvvvvvvvvv

            # ^^^^^^^^^^^^^^

            # importance sampling factor for TD(n) Q-Learning
            if on_policy:
                rho = 1
            else:
                # vvvvvvvvvvvvvvvvvvvvvvvvvv

                # ^^^^^^^^^^^^^^^^^^^^^^^^^^

            # updates
            # vvvvvvv

            # ^^^^^^^

        reward_sum += reward
        state = next_state

        # roll trace to make space for next transition at the end
        trace = np.roll(trace, shift=-1, axis=0)

        # fill with dummy transitions so we can learn from end of episode
        done_n += done
        if done_n > n:
            break

    # testing the learned policy
    state, _ = env.reset()
    test_reward_sum = 0
    while True:
        action = learned_policy.sample(state)
        next_state, reward, term, trun, _ = env.step(action)
        done = term or trun
        test_reward_sum += reward
        state = next_state
        if done:
            break

    update_plot(int(np.ceil(num_episodes / 20)))

env.close()
experiment_id = next_experiment()
print("Sampling policy and values")
plot(env, v=V, policy=sample_policy, draw_vals=True)
print("Learned policy and optimal/max values")
plot(env, v=Q.max(axis=1), policy=learned_policy, draw_vals=True)
```

SARSA(n)

Exercise

- Implement SARSA(n) by filling in the `n_step_return`, `v_target` and `q_target` in the skeleton.

  Answer:

```
# n-step targets
n_step_return = sum(gamma ** i * r for i, (_, _, r, _) in enumerate(trace))
v_target = n_step_return + gamma ** (n + 1) * V[next_state]
q_target = n_step_return + gamma ** (n + 1) * Q[next_state, next_action]
```

- Run this on the 8x8 environment and compare different values of n in terms of performance (how quickly the agent learns) and run time (roughly how fast/slow everything is running). What effects do you observe?

  Generally, the larger n the better the agent's learning performance i.e. it jumps to an optimal (up to exploration) policy earlier. This is because observed rewards "travel" n steps at a time. However, larger n also considerably slow down the entire process (as longer traces have to be maintained and evaluated).

Q-Learning

For Q-Learning there is a problem because we want to evaluate a different policy (π^*) than the one used for sampling (π). That means that the n-step return was sampled with the "wrong" policy (π^* might never take some of the actions sampled from π) and so is not representative for π^*. This bias can be corrected by adding an *importance sampling* factor (a general technique in Monte Carlo methods to correct for sampling from a "wrong" distribution) to the learning rate α

$$\begin{aligned}\rho_t &= \frac{\pi^*(a_t \,|\, s_t)\,\pi^*(a_{t+1} \,|\, s_{t+1})\;\ldots\;\pi^*(a_{t+n} \,|\, s_{t+n})}{\pi(a_t \,|\, s_t)\,\pi(a_{t+1} \,|\, s_{t+1})\;\ldots\;\pi(a_{t+n} \,|\, s_{t+n})} = \frac{\prod_{k=0}^{n} \pi^*(a_{t+k} \,|\, s_{t+k})}{\prod_{k=0}^{n} \pi(a_{t+k} \,|\, s_{t+k})} \\ \alpha_t &= \alpha\rho_t \;.\end{aligned}$$

Exercise

- Implement TD(n) Q-Learning by
  - using different `sample_policy` and `learned_policy`
  - computing the importance sampling factor ρ (`policy[s][a]` is giving you the probability of taking action `a` in state `s`).
  - modifying the updates accordingly.

  Compute importance sampling factor and change updates:

```
# importance sampling factor
rho = np.prod([learned_policy[s][a] / sample_policy[s][a] for s, a, _, _ in trace])
# updates
s, a, _, _ = trace[0]
V[s] += alpha * rho * (v_target - V[s])
Q[s, a] += alpha * rho * (q_target - Q[s, a])
```

- Test TD(n) Q-Learning with different values for n and exploration in the sampling policy on the 8x8 environment. *Hint:* Use some small non-zero value for `epsilon` in the learned policy to make sure the importance sampling factors are not (almost) all zero. What do you observe?

  TD(n) Q-Learning is less stable compared to SARSA(n). With `epsilon=1` (i.e. full exploration) in the sampling policy, even a value of $n = 1$ results in instabilities and the agent does not learn a good policy. This is due to the very different probabilities between the two policies, which increases the noise in the TD targets. Additionally, full exploration does not provide many successful runs ending with a reward to learn from. Using smaller learning rates `alpha` can help increase stability.

  With intermediate values (e.g. `epsilon=0.5`) in the sampling policy, learning is more reliable for small values of n. However, for larger n ($\approx$10) learning success is only intermittent: it starts quickly but then degenerates again. This is again because of the noise due to the differences in the two policies. Moreover, since the sampling policy is "cheating" by using the Q values from the learned policy, it also is affected by errors due to the noise.