EXERCISE 10
Monte-carlo methods and TD learning

Tue Herlau
tuhe@dtu.dk

12 April, 2024

Objective: Although the dynamic programming tools to solve a *known* MDP by evaluating the policy and optimizing it, many real world problems come with (partly) *unknown* dynamics. Today we'll therefore learn how to use model-free prediction to estimate the value function of an *unknown* MDP using Monte Carlo methods and Temporal-Difference Learning. (48 lines of code)
Exercise code: `https://lab.compute.dtu.dk/02465material/02465students.git`
Online documentation: 02465material.pages.compute.dtu.dk/02465public/exercises/ex10.html

Contents

1 Conceptual question: First and every visit

☞ First visit and every visit Monte-Carlo will both estimate the value-function accurately given enough episodes, however, this is by no means obvious since it is easy to construct examples where every-visit is clearly biased. I found this confusing, since I felt I must

have misunderstood what every and first-visit Monte Carlo did, and I found it helpful to work through an example that shows they indeed agree. Asides that, there are a few other reasons to stick through the calculation:

- First and every-visit Monte Carlo are not the most fun methods to implement due to bookkeeping, so therefore working with them theoretically is perhaps better.
- Our subsequent methods will resemble every-visit in that they perform an update every time a state is visited so therefore this is a fairly important result.
- It provides good practice in the terminology in [SB18]
- The derivation which uses the geometric series resemble other arguments in [SB18].

1.1 The single-state example

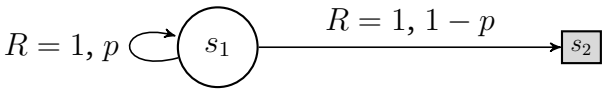


Figure 1: A simple MRP with one non-terminal state s_1 and one terminal state. With probability p the process stay in s_1 and with probability $1 - p$ it jumps to s_2, and in each jump it gets a reward of $R_t = 1$.

Consider the MRP shown in fig. 1.

- There is a single normal state s_1 and one terminal state s_2
- In each step, the agent obtains a reward of $R_{t+1} = 1$
- With probability p, the agent stay in s_1, and with probability $1 - p$ the environment terminates

This means that for an episode of length $T = 3$, the states and rewards are:

$$S_0 = s_1, S_1 = s_1, S_2 = s_1, S_3 = s_2, \quad \text{and} \quad R_1 = R_2 = R_3 = 1.$$

Note that we visited s_1 exactly T times. We are interested in estimating the value-function using first and every visit Monte-Carlo, and in both cases the value function is estimated as an average of the returns (see [SB18, Algorithm 5.1]). We write this average as:

$$V^{\text{first}}(s_1) = \text{Average}(\text{Returns}(s_1)) = \frac{S^{\text{first}}(s_1)}{N^{\text{first}}(s_1)} \tag{1}$$

Where $N^{\text{first}}(s_1)$ are the number of returns in the average and $S^{\text{first}}(s_1)$ is the sum of the returns. Similarly we define the estimate for every visit as:

$$V^{\text{every}}(s_1) = \frac{S^{\text{every}}(s_1)}{N^{\text{every}}(s_1)}$$

Thus, when we analyze the methods, all we need to keep track of is how S and N are updated.

When solving the following problems, the only tool you need is the famous Geometric series[1]:

$$1 + a + a^2 + a^3 + \cdots + a^n = \sum_{k=0}^{n} a^k = \frac{1 - a^{n+1}}{1 - a}$$

(a.) For a single episode of length T, show that the accumulated reward using first-visit Monte Carlo is (see Todays lecture)

$$S^{\text{first}}(s_1) = \frac{1 - \gamma^T}{1 - \gamma}, \quad N^{\text{first}}(s_1) = 1$$

(b.) For a single episode of length T, using the every-visit estimator, show that the estimated return is

$$S^{\text{every}}(s_1) = \frac{1}{1 - \gamma}\left[T - \gamma\frac{1 - \gamma^T}{1 - \gamma}\right], \quad N^{\text{every}}(s_1) = T$$

Hint: How many times do we visit s_1?

(c.) How can we see that every-visit is biased when $m = 1$?

(d.) Suppose we record m episodes of length $T_1, T_2, \ldots, T_m$. What are the values of $S^{\text{first}}(s_1)$ and $N^{\text{first}}(s_1)$ in this case?

(e.) Suppose we record m episodes of length $T_1, T_2, \ldots, T_m$. What are the values of $S^{\text{every}}(s_1)$ and $N^{\text{every}}(s_1)$ in this case?

(f.) challenge We want to show that $\frac{S^{\text{first}}(s_1)}{N^{\text{first}}(s_1)} \approx \frac{S^{\text{every}}(s_1)}{N^{\text{every}}(s_1)}$ when m is large. To do that we need a few more details about the problem. You can assume (and possibly, derive?) that

$$p(T) = (1 - p)p^{T-1}, \quad T = 1, 2, \ldots,$$

Assume (or show!) that $\mathbb{E}[T] = \frac{1}{1-p}$. Use this to argue why $\frac{m}{N^{\text{every}}(s_1)} \approx 1 - p$ when m is large.

(g.) Assume (or show!) that $\mathbb{E}[\gamma^T] = \gamma\frac{1-p}{1-\gamma p}$. Use this to argue that $\frac{S^{\text{first}}(s_1)}{N^{\text{first}}(s_1)} = \frac{S^{\text{every}}(s_1)}{N^{\text{every}}(s_1)}$ when $m \to \infty$.

2 Monte-Carlo evaluation (`mc_evaluate.py`)

The first task we will consider is how to evaluate a policy π using the MC method, i.e. estimate the value function $v_\pi(s)$ using a fixed policy. The only difference from the learning setting is that the agent does not try to change the policy π.

To implement our agents, I have adopted the `Agent` class to allow us to fix the policy. If this argument is not set, the agent will take random actions. Our basic policy-evaluation agent looks like this:

[1] `https://en.wikipedia.org/wiki/Geometric_series`.

```python
# rl_agent.py
class ValueAgent(TabularAgent):
    def __init__(self, env, gamma=0.95, policy=None, v_init_fun=None):
        self.env = env
        self.policy = policy  # policy to evaluate
        """ self.v holds the value estimates.
        Initially v[s] = 0 unless v_init_fun is given in which case v[s] =
        ↪ v_init_fun(s). """
        self.v = defaultdict2(float if v_init_fun is None else v_init_fun)

    def pi(self, s, k, info=None):
        return TabularAgent.pi(self, s, k, info) if self.policy is None else
        ↪ self.policy(s)
```

The only interesting thing is that it defines a dictionary `self.v` (with default value 0) which should be used to store the value function v_π.

The easiest way to implement the Monte-Carlo policy evaluation method [SB18, Section 5.1] is to keep track of the sum of returns and number of times a state has been visited similar to eq. (1). We do this using two dictionaries:

```python
# mc_evaluate.py

class MCEvaluationAgent(ValueAgent):
    def __init__(self, env, policy=None, gamma=1, alpha=None, first_visit=True,
    ↪ v_init_fun=None):
        self.episode = []
        self.first_visit = first_visit
        self.alpha = alpha
        if self.alpha is None:
            self.returns_sum_S = defaultdict(float)
            self.returns_count_N = defaultdict(float)
```

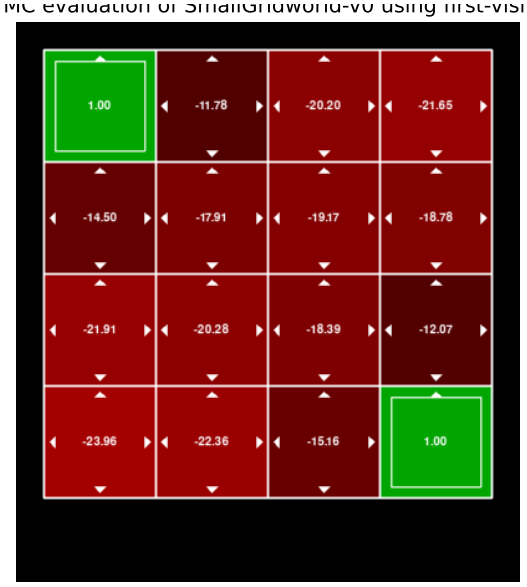
The parameter `alpha` is for the gradual implementation and for now you can assume `alpha=None`.
The list `self.episode` is for collecting an episode as `episode = [(s0, a0, r1), (s1, a1, r2), ...]`.

Problem 1 *Monte-Carlo policy evaluation*

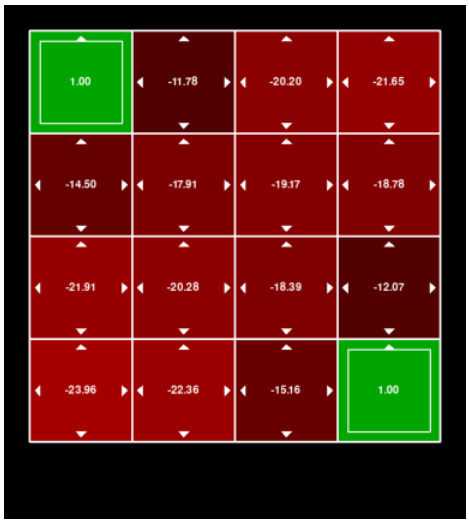
Implement the Monte-Carlo policy evaluation method [SB18, Section 5.1]. first concentrating on the first-visit setting, i.e. the code given in [SB18, Section 5.1]. To simplify the method, use the helper method which computes the returns in each state called `get_MC_returns`.

When this works, extend the code to compute every-visit MC. If you have implemented the method correctly, this can be accomplished by only changing the code in `def get_MC_return_S`.

Info: Once completed, the code should be able to evaluate the random policy. This is not very exciting as we have already seen the result once:



MC evaluation of SmallGridworld-v0 using every-visit



2.1 Examining first and every-visit

The code in the previous problem will outputs the mean of the value $v_\pi(s0)$ estimated from $m = 200$ episodes:

```
Estimated value functions v_pi(s0) for first visit -19.58
Estimated value functions v_pi(s0) for every visit -17.583955223880597
```

As an experiment, we repeat the above experiment to estimate the mean of the value function, however we use just $m = 1$ episode per run, for each run compute the mean as above and then repeat this `repeats` times. For `repeats=5000` we get the following:

```
First visit: Mean of value functions E[v_pi(s0)] after 5000 repeats
↪  -13.671696486642011
Every visit: Mean of value functions E[v_pi(s0)] after 5000 repeats -18.5902
```

Problem 2

- Why does every-visit over-estimate the mean in this case?

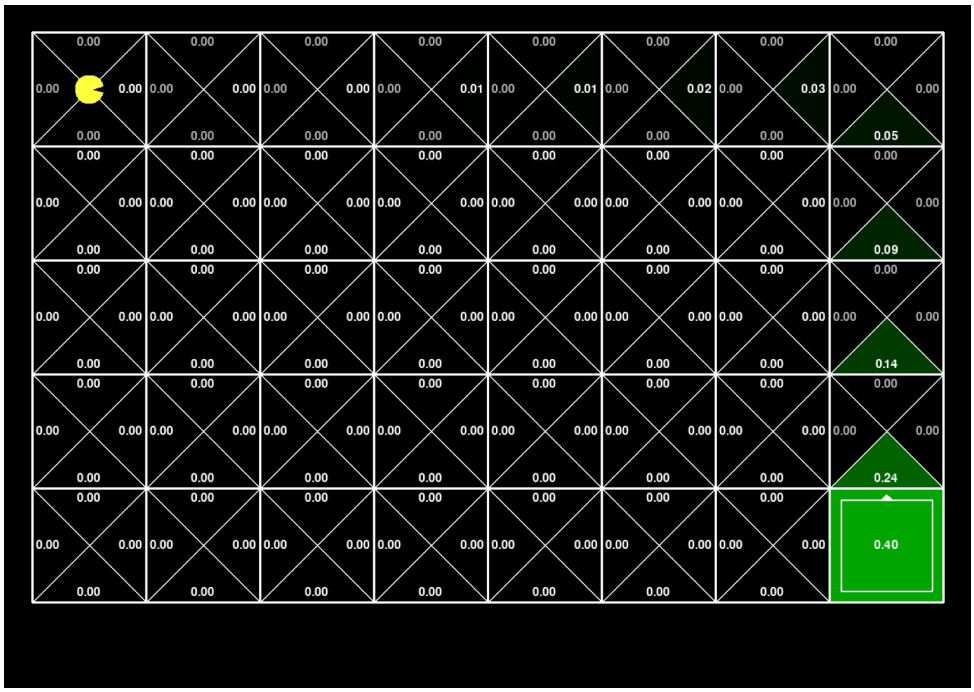


Figure 2: Monte-Carlo applied to a gridworld environment.

3 Exam question: Monte-Carlo control

We consider the familiar gridworld environment discussed in [Her24, section 4.2.4] shown in fig. 2. The agent only recieves a reward of $+1$ on completion and otherwise no reward. We train a first-visit Monte-Carlo agent on the gridworld for one episode and fig. 2 shows the resulting Q-values. What value of the discount factor γ was used?

a. $\gamma = 0.5$

b. $\gamma = 0.4$

c. $\gamma = 0.6$

d. $\gamma = 0.3$

e. Don't know.

3.1 Monte Carlo Control (`mc_agent.py`)

Next, we will build a controller using the same idea as policy evaluation. As hinted, the code you build for computing returns can be re-used; the specific version we will implement is the first-visit MC controller using ε-soft policies to get exploration as described in [SB18, Section 5.4].

As ε degrades the policy, it is important to set it quite low and we choose $\varepsilon = 0.05$ in our experiments. However, too low and the agent will not do any exploration. Similar to the evaluation-agent, we will compute the mean as:

$$Q(s, a) = \frac{S(s, a)}{N(s, a)}$$

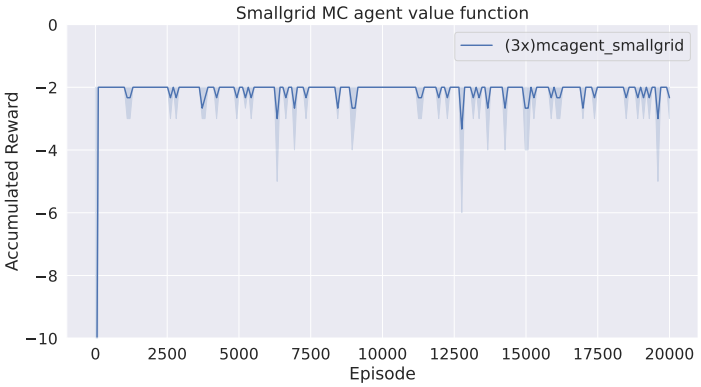
```python
# mc_agent.py
class MCAgent(TabularAgent):
    def __init__(self, env, gamma=1.0, epsilon=0.05, alpha=None, first_visit=True):
        if alpha is None:
            self.returns_sum = defaultdict(float)
            self.returns_count = defaultdict(float)
        self.alpha = alpha
        self.first_visit = first_visit
        self.episode = []
        super().__init__(env, gamma, epsilon)
```

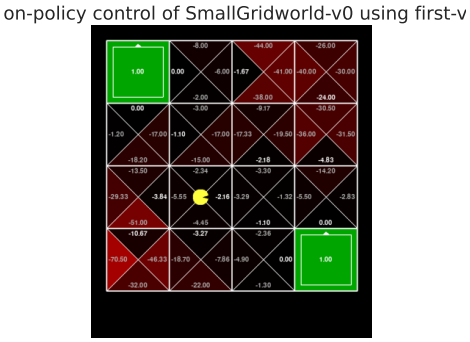
You can also assume that `alpha=None`. With these assumptions, the implementation is very nearly equal to the evaluation-agent.

Problem 3

Implement the MC on-policy Agent and test it to see if you can find the optimal (or near-optimal, since we are doing ε-greedy exploration) policy in the gridworld. Note that the method is quite variable in my experience. Why do you think we make the environment time-limited? And is this theoretically well justified given what you know about the environment?

Info: The value function of the estimated policy should look something like the following (you can compare to the previous weeks exercises). Meanwhile, the right-hand pane shows a plot of the estimated returns over time. Note there is a large deviation initially as the (bad) policy becomes stuck and can only be saved through random moves.





4 Incremental updates using α ★★

Problem 4

Update the MC Evaluation and MC Control agent to perform incremental updates as described in [SB18, Equation 6.1].

5 Temporal Difference

Monte-Carlo methods, in particular first-visit Monte-Carlo methods, estimate the value function in a given state s by considering the full future history of what happened from s onwards. The disadvantage of this approach is it requires us to actually get to the end-state before we can compute the return of s_t, and there will generally be a compounding effect of randomness in our estimate of $v_\pi(s_t)$. TD learning solves this by only considering a single-step lookahead and by assuming $v_\pi(s_{t+1})$ provides a good-enough estimate of the return following the next state s_{t+1}. This makes TD learning both extremely easy to implement, and also in a sense the opposite extreme relative to MC methods.

5.1 Implement $\mathrm{TD}(0)$ (`td0_evaluate.py`)

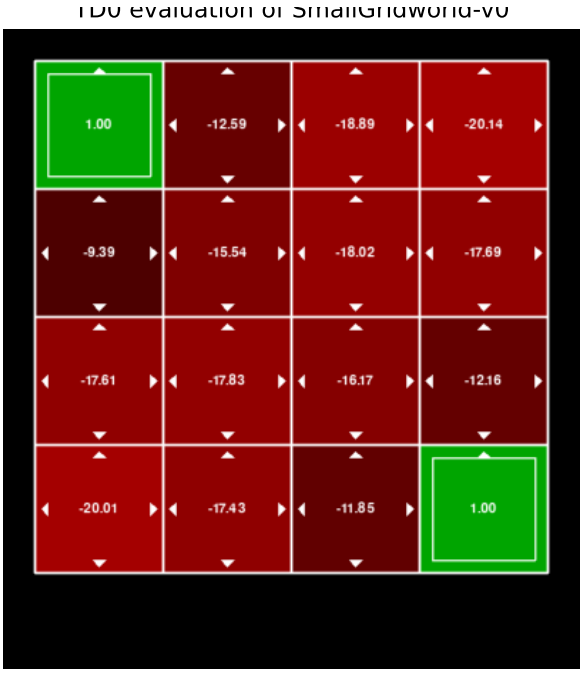
Next, we will implement the temporal difference learning method. The actual code that needs to be written will be very little, but note that the datastructure `self.v[s]` is a dictionary which defaults to zero in this case.

Problem 5

Complete the code for the TD(0) evaluation agent and, once more, use it to compute the value function for the gridworld environment using a random policy. As a test you understand what goes on: Where, exactly, is the policy specified and how is it random?

Once you have plotted the value function, compare to the MC result. Note the value function should be symmetric. What conclusions would you draw about the relative benefits of the two methods?

Info: The estimated returns should look as follows



6 Exam question: Batched TD(0) (`question_td0.py`)

In this problem, we will consider the TD(0) method described in [SB18, Section 6.1]. In particular, we will consider how TD(0) change the value function as a result of a single episode as discussed in [SB18, Example 6.1].

To this end, recall that the TD(0) algorithm, as defined in the beginning of [SB18, Section 6.1], is comprised of:

- Initializing the value function v to $v(s) = 0$
- For each episode:
  - loop over the time steps t of the episode and perform an update of v involving the current state s_t, reward r_{t+1}, discount factor γ and learning rate α

We are concerned with the last bullet point. In other words, for a given episode comprised of a list of states $(s_0, s_1, \ldots, s_T)$ and list of rewards $(r_1, r_2, \ldots, r_T)$, we can compute the full update of the value function v resulting from these observations. This is the view we will take in this exercise.

We will as usual represent the value function as a dictionary, where the keys `s` are states s, and the values `v[s]` is the value-function $v(s)$. It will be pre-initialized to zero in the test code. The states will be integers and the rewards floating point numbers.

(a.) Complete `def a_compute_deltas(v: dict, states: list, rewards: list, gamma: float)`: Given a value function v as a dictionary, lists of states and rewards corresponding to an episode, and the discount factor γ, the function should return a list comprised of the values:

$$(\delta_0, \delta_1, \ldots, \delta_{T-1})$$

where δ_t is the TD error defined in [SB18, Equation 6.5].

(b.) Complete `def b_perform_td0(v: dict, states: list, rewards: list, gamma: float, alpha: float):` Given inputs of the previously described format, as well as a learning rate α, the function should compute the TD(0) update to v resulting from a single episode and return the updated value function v (as a dictionary).

(c.) Complete `def c_perform_td0_batched(v: dict, states: list, rewards: list, gamma: float, alpha: fl` We will now consider the batched view of TD(0) as described in [SB18, Section 6.3]. In the batched version, all changes to v that arise during an episode (i.e., the factors $\alpha\delta_t$ in [SB18]) are computed as usual using the *same* value function v, but the changes are only applied to v at the *end* of the episode all at once. The function should compute the batched update and return the updated version of v in the usual format.

References

[Her24] Tue Herlau. Sequential decision making. (Freely available online), 2024.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. (Freely available online).