EXERCISE 10
Monte-carlo methods and TD learning

Tue Herlau
tuhe@dtu.dk

8 April, 2022

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. (37 lines of code)
Material: Obtain exercise material from our gitlab repository at
`https://gitlab.gbar.dtu.dk/02465material/02465students`

Contents

1 Monte Carlo prediction

Monte Carlo Methods solve the reinforcement learning problem by averaging sample returns from (actual or simulated) episodes of experience. This is often not the most effective strategy, but it is simple to implement and easy to understand why it works. In addition, more advanced methods, such as eligibility traces (which we will encounter later) can be considered as a combination of MC evaluation and the one-stage TD evaluation strategy we will consider today.

1.1 Implementation issues

In this and the remaining lectures of the course we will only use the Agent/Environment framework which we have seen many times in the course. For visualization of the various methods we will use the dark-gridworld environment which allows us to plot the value/action-value functions easily. An introduction to visualization can be found at [Her21, section 4.4.5]), but let's consider a couple of examples.

Training and visualizing environments Let's suppose we want to train an agent (in this case I have chosen the TD(0) agent) on a simple gridworld. We first import and instantiate the gridworld as follows:

```python
# td0_evaluate.py
    from irlc.gridworld.gridworld_environments import SuttonCornerGridEnvironment
    from irlc import VideoMonitor
    env = SuttonCornerGridEnvironment() # Make the gridworld environment itself
```

The gridworlds are easy to define/modify and uses the MDP classes. The full code listing is in fact simply:

```python
# gridworld_environments.py
sutton_corner_maze = [[  1, ' ', ' ', ' '],
                      [' ', ' ', ' ', ' '],
                      [' ', 'S', ' ', ' '],
                      [' ', ' ', ' ',   1]]
"""
Implement Suttons little corner-maze environment (see (SB18, Example 4.1)).
You can make an instance using:
> from irlc.gridworld.gridworld_environments import SuttonCornerGridEnvironment
> env = SuttonCornerGridEnvironment()
To get access the the mdp (as a MDP-class instance, for instance to see the states
↪  env.mdp.nonterminal_states) use
> env.mdp
You can make a visualization (allowing you to play) and train it as:
> from irlc import PlayWrapper, Agent, train
> agent = Agent()
> agent = PlayWrapper(agent, env) # allows you to play using the keyboard; omit to
↪  use agent as usual.
> train(env, agent, num_episodes=1)
"""
class SuttonCornerGridEnvironment(GridworldEnvironment):
    def __init__(self, *args, living_reward=-1, **kwargs): # living_reward=-1 means
    ↪  the agent gets a reward of -1 per step.
        super().__init__(sutton_corner_maze, *args, living_reward=living_reward,
        ↪  **kwargs)
```

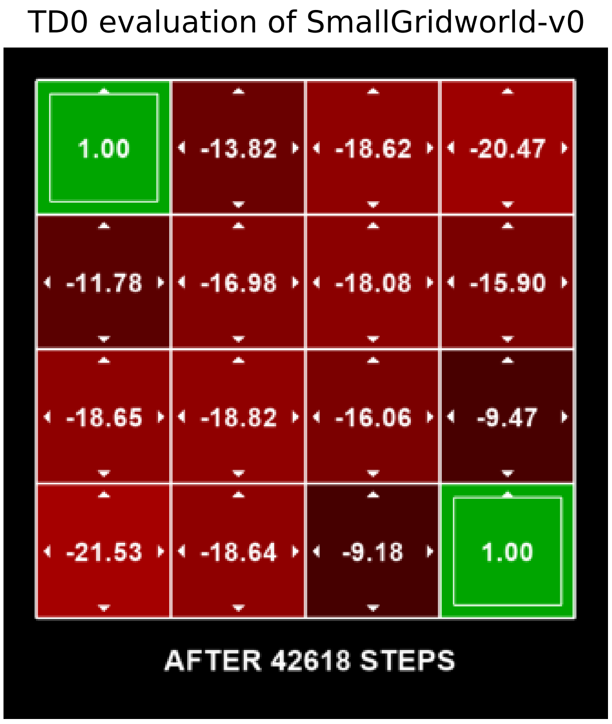


Figure 1: Example of using the visualization tools for the TD0-method

Given the gridworld, we can now create a TD0 agent and train it for 1000 episodes of data:

```python
# td0_evaluate.py
    gamma = 1
    agent = TD0ValueAgent(env, gamma=gamma, alpha=0.05) # Make a TD(0) agent
    train(env, agent, num_episodes=2000) # Train for 2000 episodes
```

Easy, but not very informative. While there are ways to plot the average reward and other statistics (see course notes and earlier exercises), we want to see the value function itself. To do this, just add a `VideoMonitor` and use the plot-function:

```python
# td0_evaluate.py
    env = VideoMonitor(env, agent=agent) # Add a video monitor, the environment will
    ↪  now show an animation
    train(env,agent,num_episodes=1) # Train for a (single) new episode
    env.plot() # Plot the current state of the environment/agent
    plt.title(f"TD0 evaluation of {envn}")
    savepdf("TD_value_random_smallgrid")
    plt.show()
```

This code will now show a small animation of the agent in the environment and save a snapshot (see fig. 1). The special command `agent_monitor_keys` is used to let the environment know this agent computes a value function; don't worry too much about this, just remember that without this command, the environment would not show the value function.

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 $v_\pi(s)$. It is important to emphasize the setting for estimating the value function is the same as the learning setting: The agent iteratively takes actions, observe consequences, and use these two to estimate v_π as quickly as possible. The only difference from the learning setting is that the agent does not try to change the policy π.

We can therefore re-use our `Agent` class, and only add a parameter to fix the the policy (the `policy` parameter), and re-use our function for training the policy. Our basic policy-evaluation agent looks like this:

```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=None):
        return self.random_pi(s) if self.policy is None else self.policy(s)
```

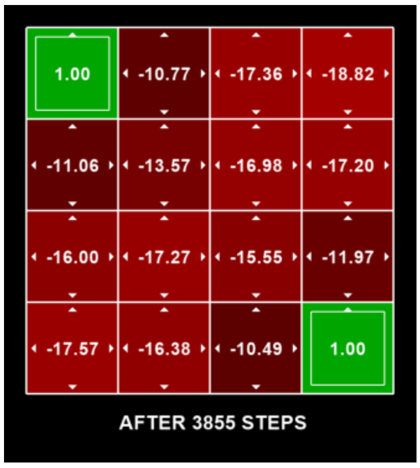
Let us put this to use by writing our basic MC Evaluation function. A slight annoyance is [SB18] has multiple versions in mind in chapter 5 and 6, and he is not always that clear which one to use. To make it clear, the version in [SB18, Section 5.1] is the first-visit no-α version, triggered when `first_visit=True, alpha=None` in our code, which we will implement first. In the code we have abstracted the function to compute returns into a separate function; this function can be re-used with slight modifications later when we write the on-policy first-visit MC agent ([SB18, Secion 5.4]).

Problem 1 *Monte-Carlo policy evaluation*

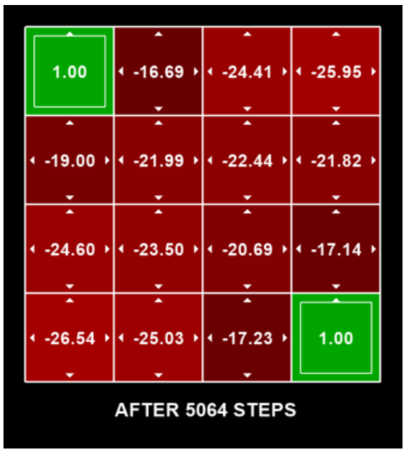
Implement the Monte-Carlo policy evaluation method, first concentrating on the first-visit, no-α setting, i.e. the code given in [SB18, Section 5.1]. When this works, extend the code to compute every-visit MC. If you have implemented the method correctly, this can be accomplished by only changing the 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 first-visit



MC evaluation of SmallGridworld-v0 using every-visit



2.1 Bias, first and every-visit

The code also outputs the mean of the value function, it is as follows, and will generally be highly variable due to the MC error:

```
Mean of value functions for first visit -16.53927182460277
Mean of value functions for every visit -16.327920532658
```

Is there a trend? I.e. on average, if we repeated the above experiment many times, would one method tend to over or under-estimate the value function? As an experiment, we repeat the above experiment to estimate the mean of the value function, however we use just 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 after 5000 repeats -11.263463221778222
Every visit: Mean of value functions after 5000 repeats -8.21328995707168
```

Problem 2

Complete the code to compute the above number, but increase the number of repeats to a large number like 10000 or 50000.

- Is there a trend? Is one method systematically over or under-estimating the value function?
- Why? Consider how the two methods work, how first-visit differs, and how this difference might occur
- What primarily determines if the method over or under-estimate the value function?
- According to [SB18], every-visit will still correctly estimate the value function asymptotically. Understand what he means, and try to explain how this statement gets around the above problem

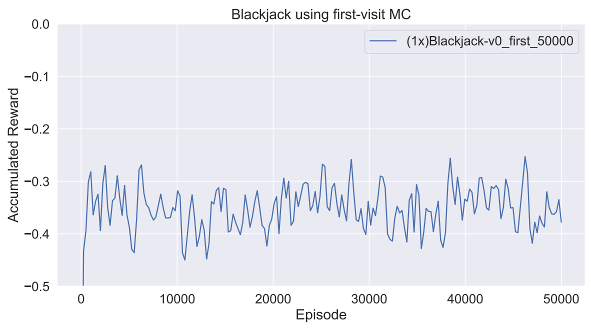
2.2 Policy evaluation for blackjack (`mc_evaluate_blackjack.py`) ✸

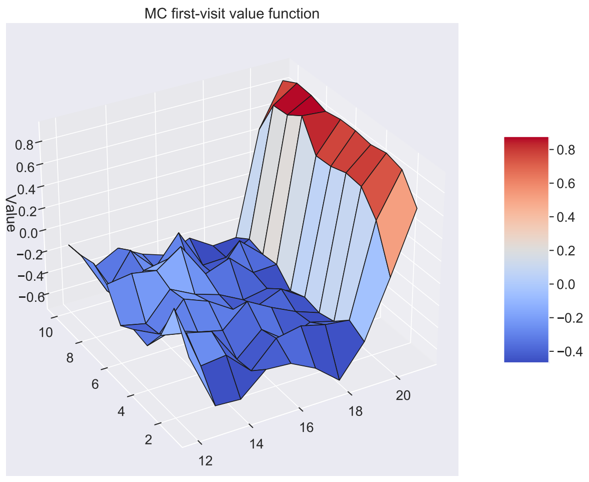
For the next task, we will consider the blackjack example from [SB18, Section 5.1]. Our first task will be to evaluate the simple policy that sticks if the player has 20 or 21.

Problem 3

Evaluate the sticks-if-20-or-more rule using first-visit MC to re-produce [SB18, Fig. 5.1]. Who is winning according to the plot of estimated returns?

Info: The estimated returns, and value function for the case the player holds an ace, looks as follows:





2.3 Monte Carlo Control (`mc_agent.py`)

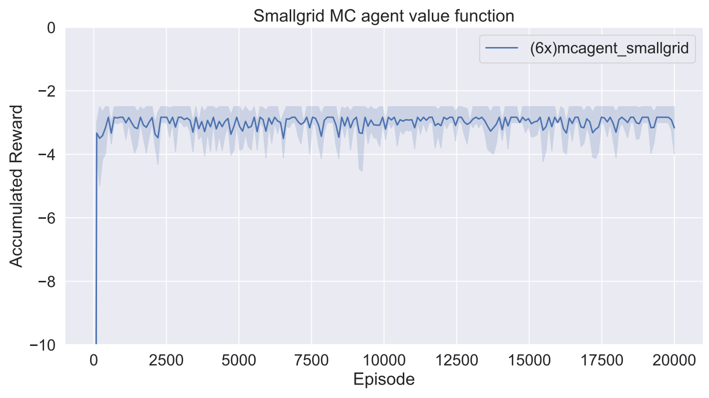
Next, we will build a controller using the same idea as policy evaluation. As hinted, the code you build for computing returns can be re-used; the specific version we will implement is the first-visit MC controller using ε-soft policies to get exploration as described in [SB18, Section 5.4].

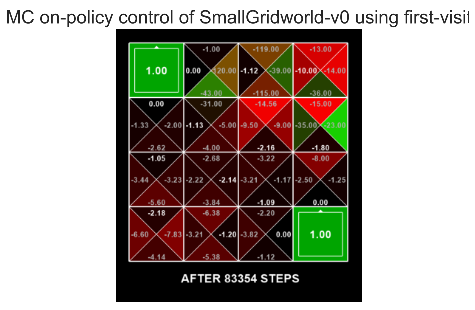
As ε degrades the policy, it is important to set it quite low and we choose $\varepsilon = 0.05$ in our experiments. However, too low and the agent will not do any exploration.

Problem 4

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.





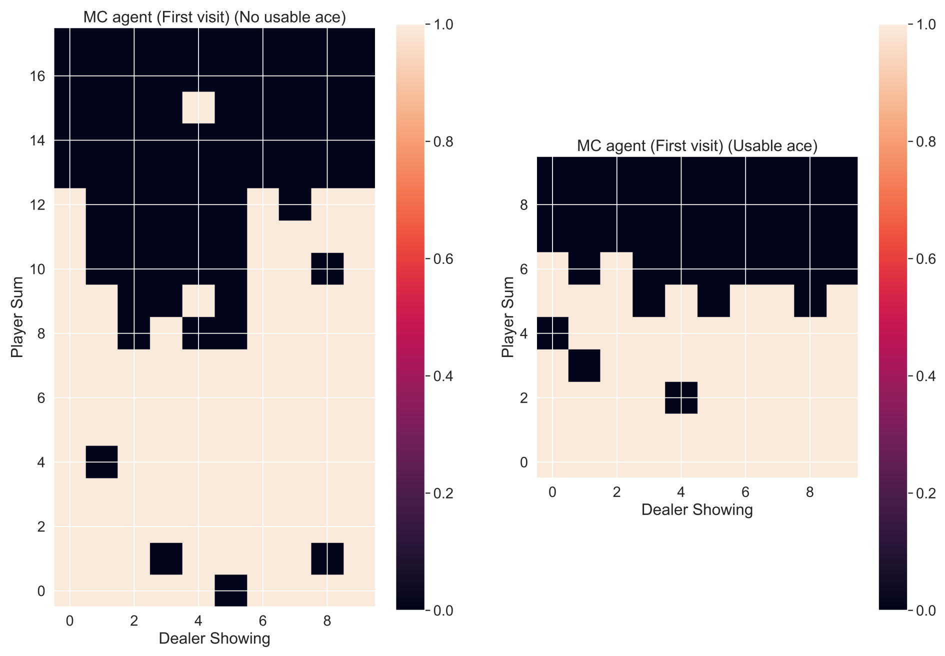
2.4 Solving blackjack (`mc_agent_blackjack.py`) ✸

Next, let's try to solve the blackjack environment, but compared to [SB18] we will not use exploring starts (since this is extra work) but rather our MC agent using ε-soft exploration.

Problem 5

Complete the script and compare your policy to the one in [SB18, Figure 5.2]. We did not quite get the same result even using twice as many samples. How do you explain our Monte-Carlo method is less efficient than the exploring-starts version?

Info: The estimated returns and the player policy looks as follows:



3 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.

3.1 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.

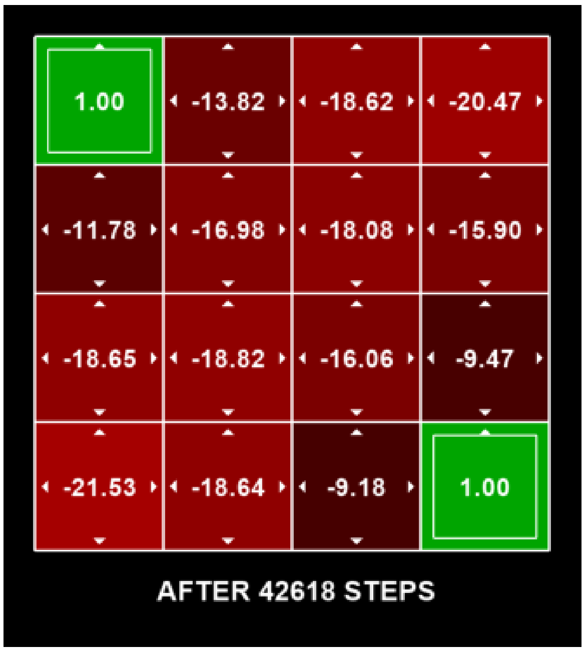
Problem 6

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

TD0 evaluation of SmallGridworld-v0



3.2 The random walk example (`random_walk_example.py`) ✶✶

We will now conisder the simple Random Walk in the Markov Reward Process (MRP) example from [SB18, Example 6.2]. Recall a MRP is a MDP with a single action, which happens to do nothing. In other words the example can be considered a comparison of how well TD and MC (first visit) learning approximates the value function as a function of the number of trajectories (episodes) without the complication of also having to learn a policy.

Reading the example, notice that Sutton describes a method MC-α which is not given in the book as an algorithm. You thus have to extend the MC code to include α, which is fortunately very easy. Take the two last lines in first-visit MC learning:

$$V(S_t) \leftarrow \text{average}\left(\text{Returns}(S_t)\right)$$

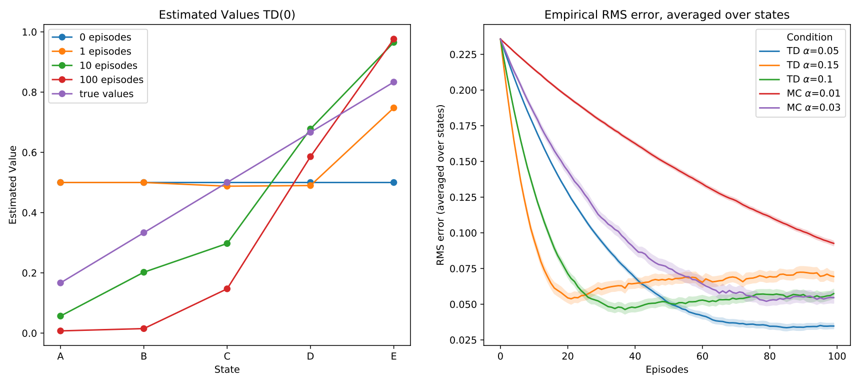
Should (as I understand it at least!) be replaced with a TD-type moving average update:

$$V(S_t) \leftarrow V(S_t) + \alpha\left[G_t - V(S_t)\right]$$

Problem 7 *Comparing TD and MC learning*

Add α update to the MC value estimation method. Then finish the script and make sure you can generate the same plot as in [SB18, Example 6.2]. Make sure you understand what is being plotted, read the example carefully, and discuss the difference between the two methods.

Info: My version looks as follows, using 95% CI:



References

[Her21] Tue Herlau. Sequential decision making. (See **02465_Notes.pdf**), 2021.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. (See **sutton2018.pdf**).