

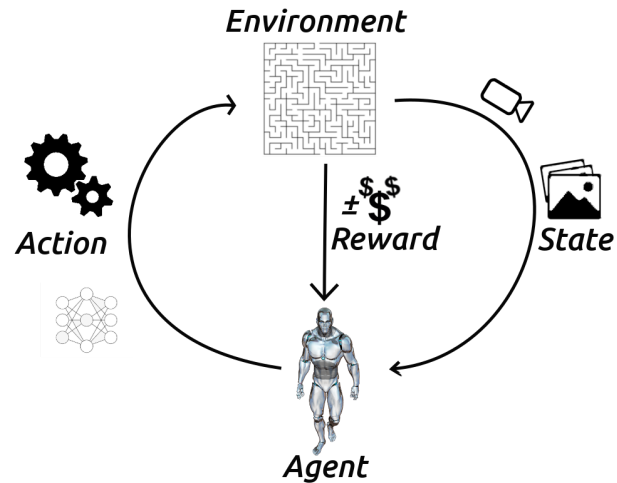


# Навчання з підкріпленням

## Лекція 4: Безмодельне передбачення

Кочура Юрій Петрович  
[iuriy.kochura@gmail.com](mailto:iuriy.kochura@gmail.com)  
[@y\\_kochura](#)

# Огляд



- Навчання з підкріпленням — це наука про те, як навчитися приймати правильні рішення
- Агент може вивчати **стратегію**, **функцію цінності** та/або **модель**
- Загальна проблема передбачає врахування **часу** та **наслідків**
- Прийняті рішення впливають на **винагороду**, **стан агента** та **стан середовища**

# Сьогодні

- Вступ до безмодельного передбачення
- Методи Монте-Карло
- Методи часових різниць (temporal difference, TD)
- TD( $\lambda$ ): Покращена оцінка усіх відвіданих станів

- Минулого разу
  - Планування за допомогою динамічного програмування
  - Розв'язок відомого МППР
- Сьогодні
  - Безмодельне передбачення
  - Оцінка функції цінності невідомого МППР

# Вступ до безмодельного передбачення

Агенти, які навчаються на зворотному зв'язку (методом проб і помилок) часто відносять до задач передбачення, тому що ми маємо оцінити функцію цінності, яка показує очікувану (середню) винагороду агента для певної стратегії. Функція цінності містить значення, які залежать від майбутнього, тому в певному сенсі ми вчимося передбачати майбутнє.

**Винагорода**  $R_t$ : скалярний сигнал, який отримує агент у якості зворотного зв'язку від середовища після виконання дії агеном. Відноситься до однокрокового сигналу: агент спостерігає за станом середовища, обирає дію та отримує сигнал винагороди. Миттєва винагорода є ключовим поняттям в RL, але це не те, що агент намагається максимізувати.

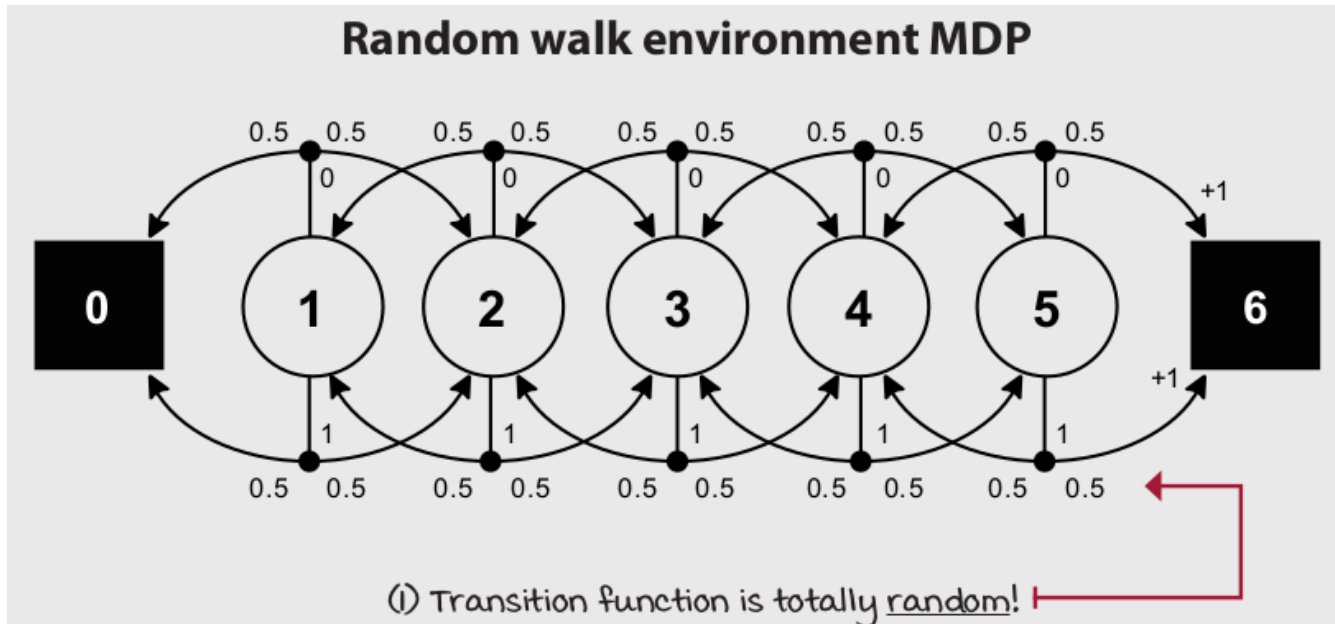
**Загальна винагорода (return)**: сумарна винагорода отримана агентом з моменту часу  $t$  з урахування знецінювання  $\gamma$ . Розраховується від будь-якого стану агента і зазвичай триває до кінця епізоду. Тобто, коли досягається **стан завершення** (terminal state), обчислення припиняється.

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

**Функція цінності**: визначає усереднену загальну винагороду:

$$\begin{aligned} v(s) &= \mathbb{E} [G_t \mid S_t = s] = \\ &= \mathbb{E} [R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \mid S_t = s] \end{aligned}$$

## Середовище випадкового блукання





# Методи Монте-Карло (МК)

- Методи МК вчаться безпосередньо з епізодів (досвіду)
- МК методи безмодельні: відсутні знання про МППР
- Методи МК навчаються з повних епізодів: без бутстрапінга
- Ми називаємо пряму вибірку епізодів Монте-Карло
- Примітка: методи МК можна застосовувати лише до епізодичних МППР
  - Усі епізоди мають бути кінцевими
- Методи МК використовують просту ідею: цінність = середня загальна винагорода

## Оцінка стратегії Монте-Карло

- Мета: вивчити  $v_\pi(s)$  з епізодів досвіду в рамках стратегії  $\pi$

$$S_t, A_t, R_{t+1}, \dots, S_T \sim \pi$$

- Загальна винагорода:

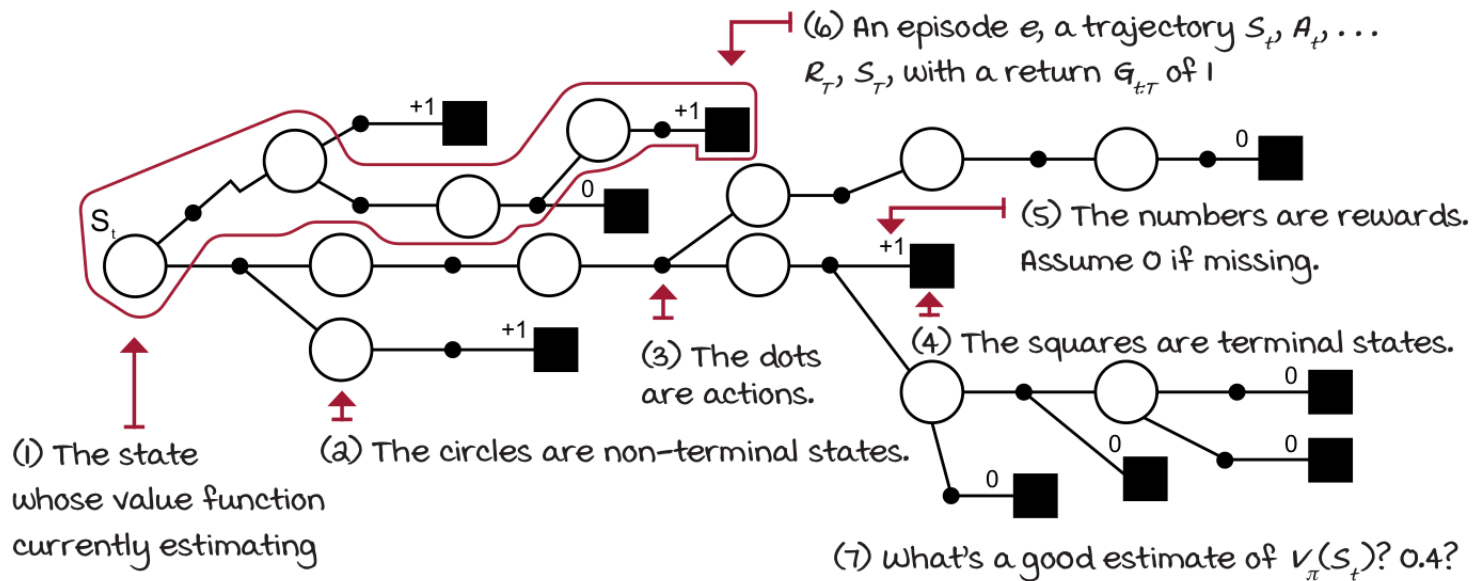
$$G_{t:T} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T$$

- Функція цінності:

$$v_\pi(s) = \mathbb{E}_\pi [G_t \mid S_t = s]$$

## Перше відвідування Монте-Карло (**First-visit Monte Carlo, FVMC**): покращення оцінок після кожного епізоду

### Monte Carlo prediction



## FVMC

1.  $v_\pi(s) = \mathbb{E}_\pi [G_t \mid S_t = s]$
2.  $G_{t:T} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T$
3.  $S_t, A_t, R_{t+1}, \dots, R_T, S_T \sim \pi$
4.  $T_T(S_t) = T_T(S_t) + G_{t:T}$
5.  $N_T(S_t) = N_T(S_t) + 1$
6.  $V_T(S_t) = \frac{T_T(S_t)}{N_T(S_t)}$
7. Якщо  $N(s) \rightarrow \infty$ , тоді  $V(s) \rightarrow v_\pi(s)$

$$V_T(S_t) = V_{T-1}(S_t) + \frac{1}{N_t(S_t)} \left[ G_{t:T} - V_{T-1}(S_t) \right]$$

$$V_T(S_t) = V_{T-1}(S_t) + \alpha_t \left[ \underbrace{G_{t:T}}_{\text{MC target}} - \overbrace{V_{T-1}(S_t)}^{\text{MC error}} \right]$$

## Кожне відвідування Монте-Карло (**Every-visit Monte Carlo, EVMC**): інший спосіб обробки відвіданих станів

Ви, напевно, помітили, що на практиці можна реалізувати два різних способи алгоритму з усереднення загальної винагороди. Це викликано тим, що одна траєкторія може містити кілька відвідувань одного і того ж стану. У цьому випадку, чи варто розраховувати загальну винагороду після кожного з цих відвідувань незалежно, а потім включити всі ці значення до усереднення, чи ми повинні використовувати обраховану загальну винагороду лише від першого візиту до кожного стану?

Обидва підходи є робочими та мають схожі теоретичні властивості.

## Перше **vs** Кожне відвідування Монте-Карло

Передбачення МК оцінює  $v_{\pi}(s)$  як усереднене значення загальних винагород при дотриманні стратегії  $\pi$ . FVMC використовує лише одне значення загальної винагороди для одного стану протягом епізоду: загальна винагорода після першого відвідування стану. EVMC усереднює загальну винагороду для усіх відвідувань одного і того ж стану протягом епізоду.

## Історія

Ви, напевно, чули раніше термін "симуляції Монте-Карло" або "імітаційне моделювання". Методи Монте-Карло, загалом відомі з 1940-х років і є широким класом алгоритмів, які використовують випадкову вибірку для оцінок. Проте, у 1996 році вперше методи першого та кожного відвідування МК були визначені у статті [Сатіндера Сінгха](#) (Satinder Singh) та [Річарда Саттона](#) "Reinforcement Learning with Replacing Eligibility Traces".





## I SPEAK PYTHON

### Exponentially decaying schedule

```
def decay_schedule(init_value, min_value, decay_ratio, max_steps, log_start=-2, log_base=10):  
    decay_steps = int(max_steps * decay_ratio)  
    rem_steps = max_steps - decay_steps
```

← (1) This function allows you to calculate all the values for alpha for the full training process.

(2) First, calculate the number of steps to decay the values using the decay\_ratio variable.  
(3) Then, calculate the actual values as an inverse log curve. Notice we then normalize between 0 and 1, and finally transform the points to lay between init\_value and min\_value.

```
    values = np.logspace(log_start, 0, decay_steps, base=log_base, endpoint=True)[: -1]  
    values = (values - values.min()) / \  
            (values.max() - values.min())  
    values = (init_value - min_value) * values + min_value  
    values = np.pad(values, (0, rem_steps), 'edge')  
    return values
```



## I SPEAK PYTHON

### Generate full trajectories

```
def generate_trajectory(pi, env, max_steps=20):  
    done, trajectory = False, []  
    while not done:  
        state = env.reset()  
        for t in count():  
            action = pi(state)  
            next_state, reward, done, _ = env.step(action)  
            experience = (state, action, reward,  
                          next_state, done)  
            trajectory.append(experience)  
        if done:  
            break  
        if t >= max_steps - 1:  
            trajectory = []  
            break  
        state = next_state  
    return np.array(trajectory, np.object)
```

(1) This is a straightforward function. It's running a policy and extracting the collection of experience tuples (the trajectories) for off-line processing.

(2) This allows you to pass a maximum number of steps so that you can truncate long trajectories if desired.



## I SPEAK PYTHON

### Monte Carlo prediction 1/2

```
def mc_prediction(pi,
                  env,
                  gamma=1.0,
                  init_alpha=0.5,
                  min_alpha=0.01,
                  alpha_decay_ratio=0.3,
                  n_episodes=500,
                  max_steps=100,
                  first_visit=True):
```

(1) The `mc_prediction` function works for both first- and every-visit MC. The hyperparameters you see here are standard. Remember, the discount factor, `gamma`, depends on the environment.

(2) For the learning rate, `alpha`, I'm using a decaying value from `init_alpha` of 0.5 down to `min_alpha` of 0.01, decaying within the first 30% (`alpha_decay_ratio` of 0.3) of the 500 total `max_episodes`. We already discussed `max_steps` on the previous function, so I'm passing the argument around. And `first_visit` toggles between FVMC and EVMC.

```
    nS = env.observation_space.n
    discounts = np.logspace(
        0, max_steps, num=max_steps,
        base=gamma, endpoint=False)
    alphas = decay_schedule(
        init_alpha, min_alpha,
        alpha_decay_ratio, n_episodes)
```

(3) This is cool. I'm calculating all possible discounts at once. This `logspace` function for a `gamma` of 0.99 and a `max_step` of 100 returns a 100 number vector: `[1, 0.99, 0.9801, ..., 0.3697]`.

(4) Here I'm calculating all of the `alphas`!

(5) Here we're initializing variables we'll use inside the main loop: the current estimate of the state-value function `V`, and a per-episode copy of `V` for offline analysis.

```
    V = np.zeros(nS)
    V_track = np.zeros((n_episodes, nS))
```

(6) We loop for every episode. Note that we're using `'tqdm'` here. This package prints a progress bar, and it's useful for impatient people like me. You may not need it (unless you're also impatient).

```
    for e in tqdm(range(n_episodes), leave=False):
```

(7) Generate a full trajectory.

```
        trajectory = generate_trajectory(
            pi, env, max_steps)
```

(8) Initialize a visits check bool vector.

```
        visited = np.zeros(nS, dtype=np.bool)
        for t, (state, _, reward, _, _) in enumerate(
            trajectory):
```

(9) This last line is repeated on the next page for your reading convenience.



## I SPEAK PYTHON

### Monte Carlo prediction 2/2

(10) This first line is repeated on the previous page for your reading convenience.

```
for t, (state, _, reward, _, _) in enumerate(trajectory):
```

(11) We now loop through all experiences in the trajectory.

(12) Check if the state has already been visited on this trajectory, and doing FVMC.

```
if visited[state] and first_visit:
```

(13) And if so, we process the next state.

```
    continue
```

(14) If this is the first visit or we are doing EVMC, we process the current state.

```
    visited[state] = True
```

(15) First, calculate the number of steps from  $t$  to  $T$ .

(16) Then, calculate the return.

```
    n_steps = len(trajectory[t:])
```

```
    G = np.sum(discounts[:n_steps] * trajectory[t:, 2])
```

```
    V[state] = V[state] + alphas[e] * (G - V[state])
```

(17) Finally, estimate the value function.

```
    V_track[e] = V
```

(18) Keep track of the episode's  $V$ .

(19) And return  $V$ , and the tracking when done.

```
return V.copy(), V_track
```



## WITH AN RL ACCENT

### Incremental vs. sequential vs. trial-and-error

**Incremental methods:** Refers to the iterative improvement of the estimates. Dynamic programming is an incremental method: these algorithms iteratively compute the answers. They don't "interact" with an environment, but they reach the answers through successive iterations, incrementally. Bandits are also incremental: they reach good approximations through successive episodes or trials. Reinforcement learning is incremental, as well. Depending on the specific algorithm, estimates are improved on an either per-episode or per-time-step basis, incrementally.

**Sequential methods:** Refers to learning in an environment with more than one non-terminal (and reachable) state. Dynamic programming is a sequential method. Bandits are not sequential, they are one-state one-step MDPs. There's no long-term consequence for the agent's actions. Reinforcement learning is certainly sequential.

**Trial-and-error methods:** Refers to learning from interaction with the environment. Dynamic programming is not trial-and-error learning. Bandits are trial-and-error learning. Reinforcement learning is trial-and-error learning, too.

# Методи часових різниць (temporal difference, TD)

## RL WITH AN RL ACCENT

### True vs. actual vs. estimated

**True value function:** Refers to the exact and perfectly accurate value function, as if given by an oracle. The true value function is the value function agents estimate through samples. If we had the true value function, we could easily estimate returns.

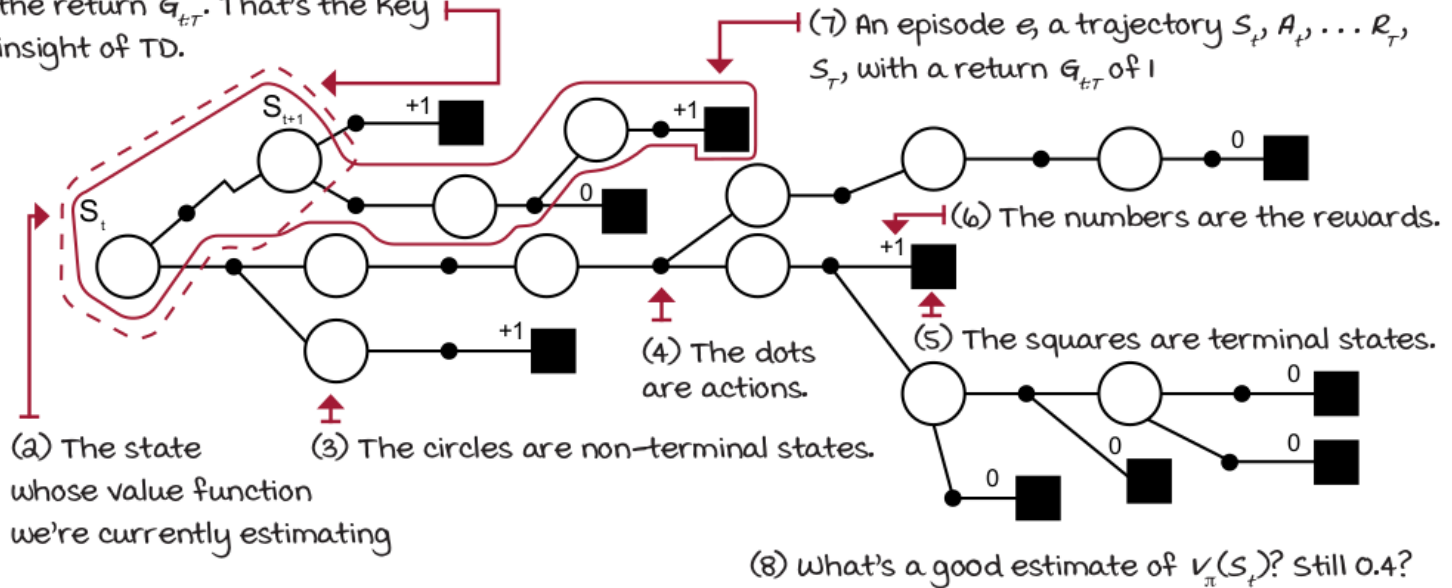
**Actual return:** Refers to the experienced return, as opposed to an estimated return. Agents can only experience actual returns, but they can use estimated value functions to estimate returns. *Actual return* refers to the full experienced return.

**Estimated value function or estimated return:** Refers to the rough calculation of the true value function or actual return. “Estimated” means an approximation, a guess. True value functions let you estimate returns, and estimated value functions add bias to those estimates.

## TD prediction

(1) This is all we need to estimate the return  $G_{t:T}$ . That's the key insight of TD.

(7) An episode  $e$ , a trajectory  $S_t, A_t, \dots, R_T, S_T$ , with a return  $G_{t:T}$  of 1





## Методи часових різниць (TD) та бутстрапінг

TD методи оцінюють  $v_{\pi}(s)$  з використанням оцінки  $v_{\pi}(s)$ . Це підхід відомий як бутстрапінг, робить здогадку з здогадки; він використовує оціночну загальну винагороду замість фактичної. Формально цей метод використовує:

$$R_{t+1} + \gamma V_t(S_{t+1})$$

для розрахунку та оцінки  $V_{t+1}(S_t)$ .

Оскільки TD використовує один крок фактичного значення загальної винагороди  $R_{t+1}$ , він усе ще працюватиме добре. Цей сигнал винагороди  $R_{t+1}$  поступово «вносить реальність» в оцінки.



## SHOW ME THE MATH

### Temporal-difference learning equations

(1) We again start from the definition of the state-value function ...  $v_{\pi}(s) = \mathbb{E}_{\pi}[G_{t:T} \mid S_t = s]$

(2) ... and the definition of the return.

$$G_{t:T} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T$$

(3) From the return, we can rewrite the equation by grouping up some terms. Check it out.

$$G_{t:T} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{T-1} R_T$$

$$= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots + \gamma^{T-2} R_T)$$

$$= R_{t+1} + \gamma G_{t+1:T} \quad (4) \text{ Now, the same return has a recursive style.}$$

(5) We can use this new definition to also rewrite the state-value function definition equation.

$$v_{\pi}(s) = \mathbb{E}_{\pi}[G_{t:T} \mid S_t = s]$$

$$= \mathbb{E}_{\pi}[R_{t+1} + \gamma G_{t+1:T} \mid S_t = s]$$

$$= \mathbb{E}_{\pi}[R_{t+1} + \gamma v_{\pi}(S_{t+1}) \mid S_t = s] \quad (6) \text{ And because the expectation of the returns from the next state is the state-value function of the next state, we get this.}$$

(7) This means we could estimate the state-value function on every time step.

$$S_t, A_t, R_{t+1}, S_{t+1} \sim \pi_{t:t+1}$$

(8) We roll out a single interaction step ...

(9) ... and can obtain an estimate  $v(s)$  of the true state-value function  $v_{\pi}(s)$  a different way than with MC.

(10) The key difference to realize is we're now estimating  $v_{\pi}(s_t)$  with an estimate of  $v_{\pi}(s_{t+1})$ . We're using an estimated, not actual, return.

$$V_{t+1}(S_t) = V_t(S_t) + \alpha_t \left[ \underbrace{R_{t+1} + \gamma V_t(S_{t+1})}_{\text{TD target}} - V_t(S_t) \right] \quad \text{TD error}$$

(11) A big win is we can now make updates to the state-value function estimates  $v(s)$  every time step.



## I SPEAK PYTHON

### The temporal-difference learning algorithm

```
def td(pi,
      env,
      gamma=1.0,
      init_alpha=0.5,
      min_alpha=0.01,
      alpha_decay_ratio=0.3,
      n_episodes=500):
```

(1) td is a prediction method. It takes in a policy pi, an environment env to interact with, and the discount factor gamma.

(2) The learning method has a configurable hyperparameter alpha, which is the learning rate.

(3) One of the many ways of handling the learning rate is to exponentially decay it. The initial value is init\_alpha, min\_alpha, the minimum value, and alpha\_decay\_ratio is the fraction of episodes that it will take to decay alpha from init\_alpha to min\_alpha.

```
    nS = env.observation_space.n
    V = np.zeros(nS)
    V_track = np.zeros((n_episodes, nS))
    alphas = decay_schedule(
        init_alpha, min_alpha,
        alpha_decay_ratio, n_episodes)
```

(4) We initialize the variables needed.

(5) And we calculate the learning rate schedule for all episodes...

(6) ... and loop for n\_episodes.

```
    for e in tqdm(range(n_episodes), leave=False):
```

(7) We get the initial state and then enter the interaction loop.

```
        state, done = env.reset(), False
        while not done:
```

(8) First thing is to sample the policy pi for the action to take in state.

```
            action = pi(state)
```

(9) We then use the action to interact with the environment... We roll out the policy one step.

```
            next_state, reward, done, _ = env.step(action)
```

(10) We can immediately calculate a target to update the state-value function estimates ...

```
            td_target = reward + gamma * V[next_state] * \
                        (not done)
```

(11) ... and with the target, an error.

```
            td_error = td_target - V[state]
            V[state] = V[state] + alphas[e] * td_error
```

(12) Finally update v(s)

(13) Don't forget to update the state variable for the next iteration. Bugs like this can be hard to find!

```
            state = next_state

            V_track[e] = V
```

(14) And return the V function and the tracking variable.

```
    return V, V_track
```



## TALLY IT UP

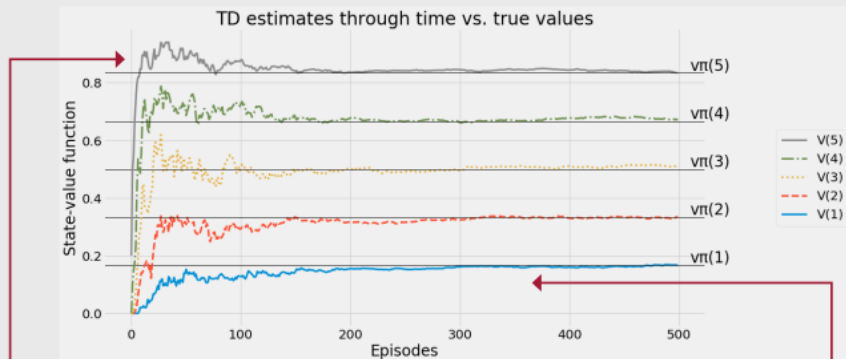
MC and TD both nearly converge to the true state-value function

(1) Here I'll show only first-visit monte Carlo prediction (FVMC) and temporal-difference learning (TD). If you head to the Notebook for this chapter, you'll also see the results for every-visit monte Carlo prediction, and several additional plots that may be of interest to you!



(2) Take a close look at these plots. These are the running state-value function estimates  $V(s)$  of an all-left policy in the random-walk environment. As you can see in these plots, both algorithms show near-convergence to the true values.

(3) Now, see the difference trends of these algorithms. FVMC running estimates are very noisy; they jump back and forth around the true values.



(4) TD running estimates don't jump as much, but they are off-center for most of the episodes. For instance  $V(5)$  is usually higher than  $v_{\pi}(5)$ , while  $V(1)$  is usually lower than  $v_{\pi}(1)$ . But if you compare those values with FVMC estimates, you notice a different trend.

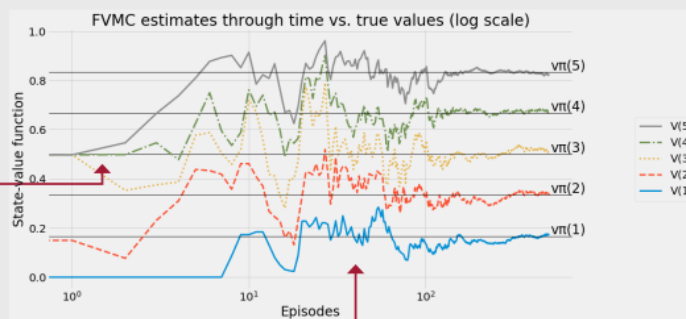
## TALLY IT UP

MC estimates are noisy; TD estimates are off-target

$$V_T(S_t) = V_{T-1}(S_t) + \alpha_t \left[ \underbrace{G_{t:T}}_{\text{MC target}} - \underbrace{V_{T-1}(S_t)}_{\text{MC error}} \right]$$

(1) If we get a close-up (log-scale plot) of these trends, you'll see what's happening. MC estimates jump around the true values. This is because of the high variance of the MC targets.

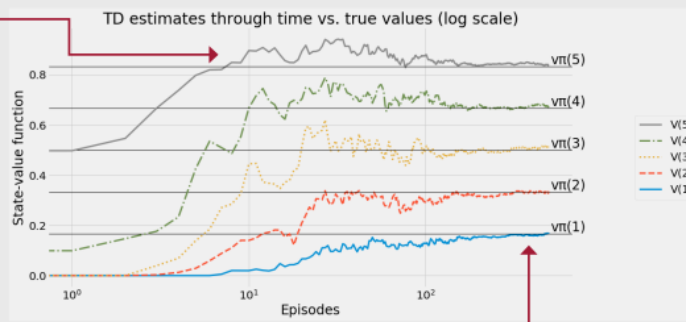
(a) A couple of pros though; first, you can see all estimates get close to their true values very early on. Also, the estimates jump around the true values.



$$V_{t+1}(S_t) = V_t(S_t) + \alpha_t \left[ \underbrace{G_{t:t+1}}_{\text{TD target}} - \underbrace{V_t(S_t)}_{\text{TD error}} \right]$$

(3) TD estimates are off-target most of the time, but they're less jumpy. This is because TD targets are low variance, though biased. They use an estimated return for target.

(4) The bias shows, too. In the end, TD targets give up accuracy in order to become more precise. Also, they take a bit long before estimates ramp up, at least in this environment.





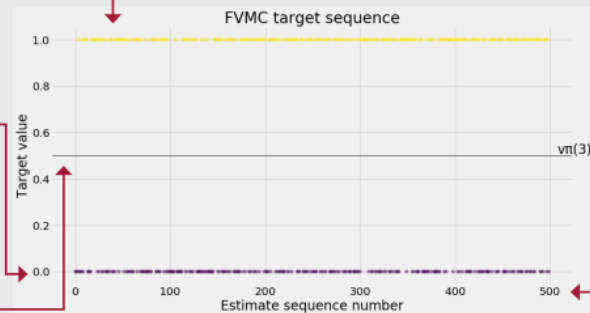
## TALLY IT UP

MC targets high variance; TD targets bias

$$G_{t:T} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T$$

(1) Here we can see the bias/variance trade-off between MC and TD targets. Remember, the MC target is the return, which accumulates a lot of random noise. That means high variance targets.

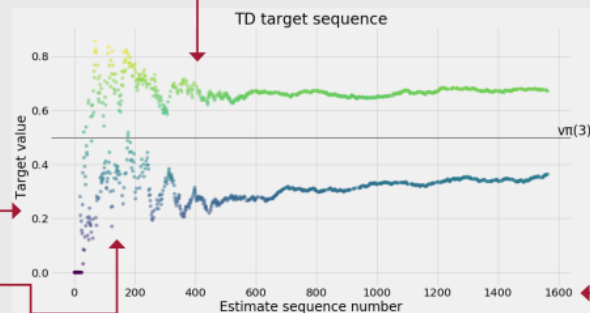
(2) These plots are showing the targets for the initial state in the RW environment. MC targets, the returns, are either 0 or 1 because the episode terminates either on the left, with a 0 return or on the right, with a 1 return, while the optimal value is 0.5!



$$G_{t:t+1} = R_{t+1} + \gamma V_t(S_{t+1})$$

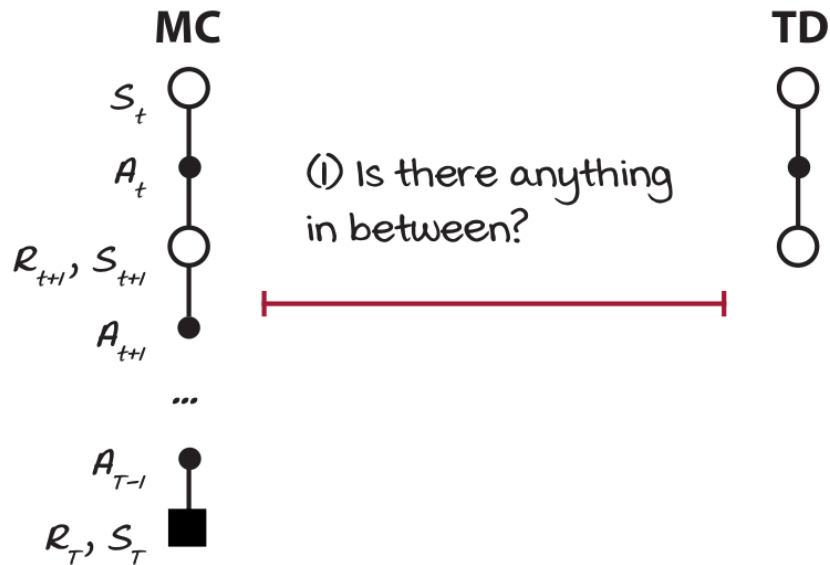
(3) TD targets are calculated using an estimated return. We use the value function to predict how much value we'll get from the next state onward. This helps us truncate the calculations and get more estimates per episode (as you can see on the x-axis, we have ~1600 estimates in 500 episodes), but because we use  $V_t(S_{t+1})$ , which is an estimate and therefore likely wrong, TD targets are biased.

(4) Here you can see the range of the TD targets is much lower, MC alternates exactly between 1 and 0, and TD jumps between approximately 0.7 and ~0.3, depending on which "next state" is sampled. But as the  $V_t(S_{t+1})$  is an estimate,  $G_{t:t+1}$  is biased, off-target, and inaccurate.



## Оцінювання функції цінності з кількох кроків

### What's in the middle?



## N-кроковый TD



### SHOW ME THE MATH

N-step temporal-difference equations

$$\xrightarrow{\quad} S_t, A_t, R_{t+1}, S_{t+1}, \dots, R_{t+n}, S_{t+n} \sim \pi_{t:t+n}$$

(1) Notice how in  $n$ -step TD we must wait  $n$  steps before we can update  $v(s)$ .

(2) Now,  $n$  doesn't have to be  $\infty$  like in MC, or 1 like in TD. Here you get to pick. In reality  $n$  will be  $n$  or less if your agent reaches a terminal state. It could be less than  $n$ , but never more.

$$\xrightarrow{\quad} G_{t:t+n} = R_{t+1} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n})$$

(3) Here you see how the value-function estimate gets updated approximately every  $n$  steps.

$$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \alpha_t \left[ \underbrace{G_{t:t+n} - V_{t+n-1}(S_t)}_{\substack{\text{n-step} \\ \text{error}}} \right]$$

(4) But after that, you can plug in that target as usual.  $\xrightarrow{\quad}$   $\underbrace{G_{t:t+n}}_{\text{n-step target}}$





## I SPEAK PYTHON

### N-step TD 1/2

```
def ntd(pi,
        env,
        gamma=1.0,
        init_alpha=0.5,
        min_alpha=0.01,
        alpha_decay_ratio=0.5,
        n_step=3,
        n_episodes=500):
```

(1) Here's my implementation of the  $n$ -step TD algorithm. There are many ways you can code this up; this is one of them for your reference.

(2) Here we're using the same hyperparameters as before. Notice `n_step` is a default of 3. That is three steps and then bootstrap, or less if we hit a terminal state, in which case we don't bootstrap (again, the value of a terminal state is zero by definition.)

```
    nS = env.observation_space.n
    V = np.zeros(nS)
    V_track = np.zeros((n_episodes, nS))
```

(3) Here we have the usual suspects.

```
    alphas = decay_schedule(
        init_alpha, min_alpha,
        alpha_decay_ratio, n_episodes)
```

(4) Calculate all alphas in advance.

(5) Now, here's a hybrid between MC and TD. Notice we calculate the discount factors, but instead of going to `max_steps` like in my MC implementation, we go to `n_step + 1` to include  $n$  steps and the bootstrapping estimate.

```
    discounts = np.logspace(
        0, n_step+1, num=n_step+1, base=gamma, endpoint=False)
```

(6) We get into the episodes loop.

```
    for e in tqdm(range(n_episodes), leave=False):
```

(7) This path variable will hold the  $n$ -step-most-recent experiences. A partial trajectory.

```
        state, done, path = env.reset(), False, []
```

(8) We're going until we hit done and the path is set to none. You'll see soon.

```
        while not done or path is not None:
```

(9) Here, we're "popping" the first element of the path.

```
            path = path[1:]
```

(10) This line repeats on the next page.

```
            while not done and len(path) < n_step:
```



## N-stepTD 2/2

(11) Same. Just for you to follow the indentation.

```

while not done and len(path) < n_step:
    (12) This is the interaction block. We're basically collecting experiences until we hit done or the length of the path is equal to n_step.
    action = pi(state)
    next_state, reward, done, _ = env.step(action)
    experience = (state, reward, next_state, done)
    path.append(experience)
    state = next_state
    if done:
        break
    (13) n here could be 'n_step' but it could also be a smaller number if a terminal state is in the 'path.'
    n = len(path)
    (14) Here we're extracting the state we're estimating, which isn't state.
    est_state = path[0][0]

    (15) rewards is a vector of all rewards encountered from the est_state until n.
    rewards = np.array(path[:n, 1])

    (16) partial_return is a vector of discounted rewards from est_state to n.
    partial_return = discounts[:n] * rewards

    (17) bs_val is the bootstrapping value. Notice that in this case next state is correct.
    bs_val = discounts[-1] * V[next_state] * (not done)

    (18) ntd_target is the sum of the partial return and bootstrapping value.
    ntd_target = np.sum(np.append(partial_return, bs_val))

    (19) This is the error, like we've been calculating all along.
    ntd_error = ntd_target - V[est_state]

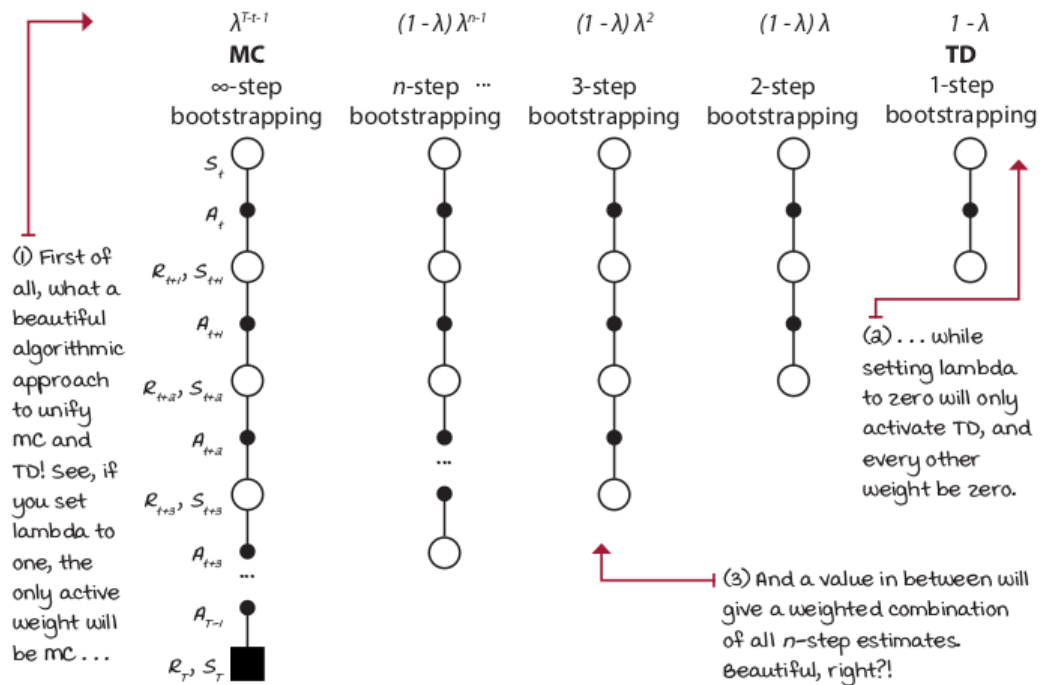
    (20) The update to the state-value function
    V[est_state] = V[est_state] + alphas[e] * ntd_error

    (21) Here we set path to None to break out of the episode loop, if path has only one experience and the done flag of that experience is True (only a terminal state in path.)
    if len(path) == 1 and path[0][3]:
        path = None
        V_track[e] = V
    return V, V_track
    (22) We return V and V_track as usual.

```

TD( $\lambda$ ): Покращена оцінка  
усіх відвіданих станів

## Generalized bootstrapping





## SHOW ME THE MATH

### Forward-view TD( $\lambda$ )

(1) Sure, this is a loaded equation; we'll unpack it here. The bottom line is that we're using all  $n$ -step returns until the final step  $T$ , and weighting it with an exponentially decaying value.

$$G_{t:T}^{\lambda} = (1 - \lambda) \underbrace{\sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n}}_{\text{Sum of weighted returns from 1-step to T-1 steps}} + \underbrace{\lambda^{T-t-1} G_{t:T}}_{\text{Weighted final return (T)}}$$

(2) The thing is, because  $T$  is variable, we need to weight the actual return with a normalizing value so that all weights add up to 1.

$$G_{t:t+1} = R_{t+1} + \gamma V_t(S_{t+1}) \quad (3) \text{ All this equation is saying is that we'll calculate the one-step return and weight it with the following factor } \dots \rightarrow 1 - \lambda$$

$$G_{t:t+2} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2}) \quad (4) \dots \text{ and also the two-step return and weight it with this factor. } \rightarrow (1 - \lambda)\lambda$$

$$G_{t:t+3} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 V_{t+2}(S_{t+3}) \quad (5) \text{ Then the same for the three-step return and this factor. } \rightarrow (1 - \lambda)\lambda^2$$

$$G_{t:t+n} = R_{t+1} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}) \quad (6) \text{ You do this for all } n\text{-steps } \dots \rightarrow (1 - \lambda)\lambda^{n-1}$$

$$G_{t:T} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-t-1} R_T \quad (7) \dots \text{ until your agent reaches a terminal state. Then you weight by this normalizing factor. } \rightarrow \lambda^{T-t-1}$$

(8) Notice the issue with this approach is that you must sample an entire trajectory before you can calculate these values.

(9) Here you have it,  $v$  will become available at time  $T$ ...

$$S_t, A_t, R_{t+1}, S_{t+1}, \dots, R_T, S_T \sim \pi_{t:T}$$

$$V_T(S_t) = V_{T-1}(S_t) + \alpha_t \left[ \underbrace{G_{t:T}^{\lambda}}_{\text{\lambda-return}} - \underbrace{V_{T-1}(S_t)}_{\text{\lambda-error}} \right] \quad (10) \dots \text{ because of this.}$$



## I SPEAK PYTHON

The TD( $\lambda$ ) algorithm, a.k.a. backward-view TD( $\lambda$ )

```

def td_lambda(pi,
              env,
              gamma=1.0,
              init_alpha=0.5,
              min_alpha=0.01,
              alpha_decay_ratio=0.3,
              lambda_=0.3,
              n_episodes=500):
    nS = env.observation_space.n
    V = np.zeros(nS)
    V_track = np.zeros((n_episodes, nS))
    E = np.zeros(nS)
    alphas = decay_schedule(
        init_alpha, min_alpha,
        alpha_decay_ratio, n_episodes)

    (5) Here we enter the episode loop.
    for e in tqdm(range(n_episodes), leave=False):
        E.fill(0)
        state, done = env.reset(), False

        while not done:
            action = pi(state)
            next_state, reward, done, _ = env.step(action)

            (9) we first interact with the environment for one step and get the experience tuple.
            (10) Then, we use that experience to calculate the TD error as usual.
            td_target = reward + gamma * V[next_state] * \
                        (not done)
            td_error = td_target - V[state]

            (11) We increment the eligibility of state by 1.
            E[state] = E[state] + 1
            V = V + alphas[e] * td_error * E
            E = gamma * lambda_ * E

            (12) And apply the error update to all eligible states as indicated by E.
            (13) We decay E...
            state = next_state
            V_track[e] = V
        return V, V_track

```

(1) The method `td_lambda` has a signature very similar to all other methods. The only new hyperparameter is `lambda_` (the underscore is because `lambda` is a restricted keyword in Python).

(2) Set the usual suspects.

(3) Add a new guy: the eligibility trace vector.

(4) Calculate alpha for all episodes.

(6) Set `E` to zero every new episode.

(7) Set initial variables.

(8) Get into the time step loop.

(9) we first interact with the environment for one step and get the experience tuple.

(10) Then, we use that experience to calculate the TD error as usual.

(11) We increment the eligibility of state by 1.

(12) And apply the error update to all eligible states as indicated by `E`.

(13) We decay `E`...

(14) ... and continue our lives as usual.



## TALLY IT UP

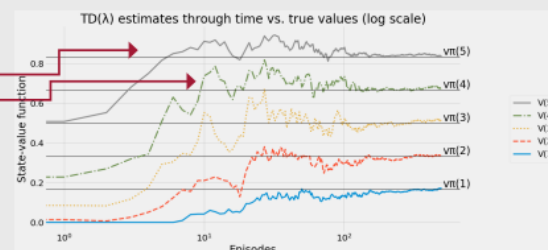
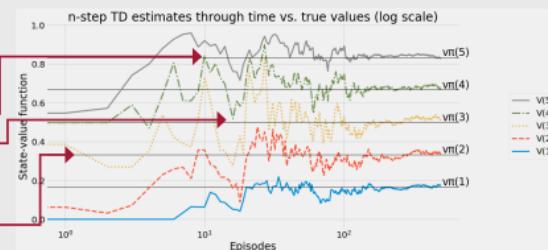
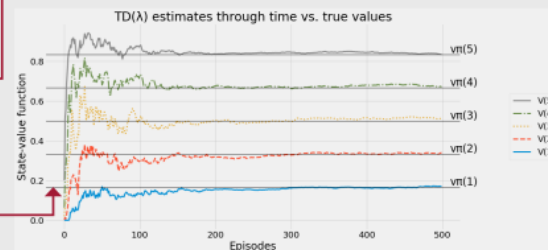
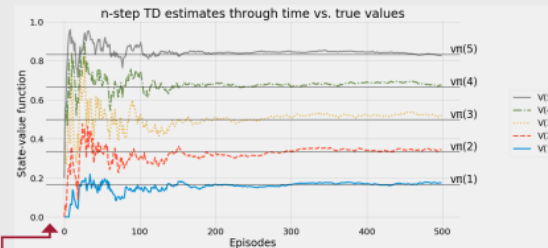
Running estimates that  $n$ -step TD and TD( $\lambda$ ) produce in the RW environment

(1) I think the most interesting part of the differences and similarities of MC, TD,  $n$ -step TD, and TD( $\lambda$ ) can be visualized side by side. For this, I highly recommend you head to the book repository and check out the corresponding Notebook for this chapter. You'll find much more than what I've shown you in the text.

(2) But for now I can highlight that  $n$ -step TD curves are a bit more like MC: noisy and centered, while TD( $\lambda$ ) is a bit more like TD: smooth and off-target.

(3) When we look at the log-scale plots, we can see how the high variance estimates of  $n$ -step TD (at least higher than TD( $\lambda$ ) in this experiment), and how the running estimates move above and below the true values, though they're centered.

(4) TD( $\lambda$ ) values aren't centered, but are also much smoother than MC. These two are interesting properties. Go compare them with the rest of the methods you've learned about so far!



Демо



Кінець

# Література

- David Silver, [Lecture 4: Model-Free Prediction](#)