EECE-6822 Machine Learning

Reinforcement Learning

Cris Ababei
Dept. of Electrical and Computer Engineering
MARQUETTE UNIVERSITY
BE THE DIFFERENCE.

1

PART 1
Reinforcement Learning

[***B3-Geron***] Aurelien Geron, Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, O'Reilly, 2022.
[***B3-Raschka***] Sebastian Raschka, Yuxi Liu, and Vahid Mirjalili, Learning with PyTorch and Scikit-Learn: Develop machine learning and deep learning models with Python, Packt Publishing, 2022.

2

First Intro

3

3

RL and Machine Learning

1. Supervised Learning (error correction)
   - learning approaches to *regression & classification*
   - learning from examples, learning from a teacher
2. Unsupervised Learning
   - learning approaches to *dimensionality reduction, density estimation, recoding data based on some principle, etc.*
3. Reinforcement Learning
   - learning approaches to *sequential decision making*
   - learning from a critic, learning from delayed reward

4

4

Reinforcement Learning (RL)

- RL - "science of decision making" or "the optimal way of making decisions"
  - When an infant plays, waves its arms, it has no explicit teacher, but it does have a direct sensorimotor connection to its environment. Exercising this connection produces a wealth of information about cause and effect, about consequences of actions, and about what to do in order to achieve goals.
- RL is centered around the concept of **learning by interaction**.
- RL - a branch of AI where an **agent** (typically a software program) gradually learns to make decisions intelligently through interaction with its environment.
- With RL, the **model** (also called an **agent**) interacts with its environment, and by doing so generates a sequence of interactions that are together called an **episode**.

5

5

Learning to Optimize Rewards

In reinforcement learning, a software *agent* makes *observations* and takes *actions* within an *environment*, and in return it receives *rewards* from the environment. Its objective is to learn to act in a way that will maximize its expected rewards over time. If you don't mind a bit of anthropomorphism, you can think of positive rewards as pleasure, and negative rewards as pain (the term "reward" is a bit misleading in this case). In short, the agent acts in the environment and learns by trial and error to maximize its pleasure and minimize its pain.

For example, imagine that we want to teach a computer to play the game of chess and win against human players. The labels (rewards) for each individual chess move made by the computer are not known until the end of the game, because during the game itself, we don't know whether a particular move will result in winning or losing that game. Only right at the end of the game is the feedback determined. That feedback would likely be a positive reward given if the computer won the game because the agent had achieved the overall desired outcome; and vice versa, a negative reward would likely be given if the computer had lost the game.

6

6

RL essence

- In RL, we cannot or do not teach an agent, computer, or robot **how** to do things.
- We can only specify **what** we want the agent to achieve. Then, based on the outcome of a particular trial, we can determine rewards depending on the agent's success or failure.
- This makes RL very attractive for decision making in complex environments, especially when the problem-solving task requires a series of steps, which are unknown, or hard to explain, or hard to define.

Example Applications

- Autonomous Vehicles
- Robotics
- Financial Trading
- Game Playing
- Precision Healthcare Optimization
- Advanced Natural Language Processing for Diverse Languages
- Resilient Supply Chain Optimization
- Eco-Friendly Smart Grid Management
- Multimodal Learning for Robotic Assistants

Examples

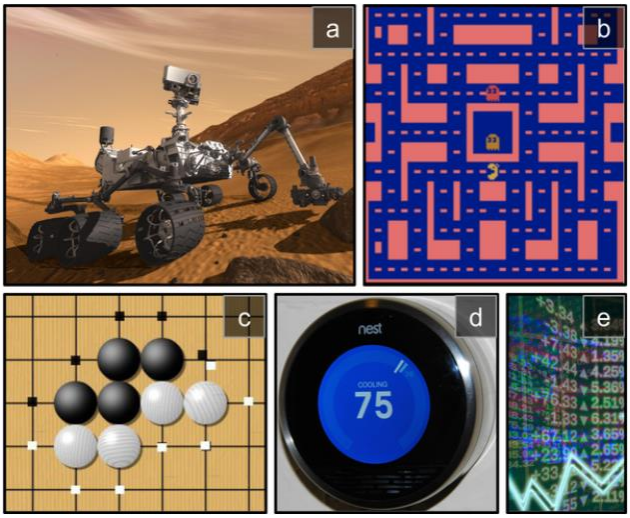


Figure 18-1. Reinforcement learning examples: (a) robotics, (b) Ms. Pac-Man, (c) Go player, (d) thermostat, (e) automatic trader[5]

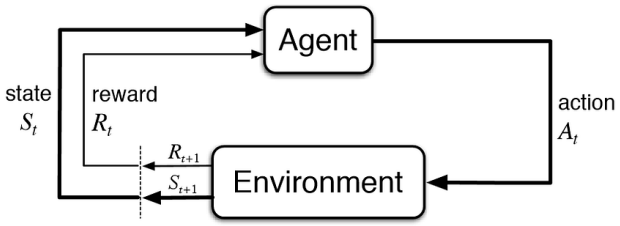
See more applications at:
https://kenndanielso.github.io/mlrefined/blog_posts/18_Reinforcement_Learning_Foundations/18_1_Fundamentals_of_reinforcement_learning.html

9

9

Key Idea

- An **environment** which **represents the outside world** to the agent
- An **agent** that **takes actions**, **receives observations** from the environment that consists of:
  - {a **Reward** for his action, information of his **New State**}
- That reward informs the agent of how good or bad was the taken action
- The observation tells him what is his next state in the environment.
- The agent tries to figure out the best actions to take or the optimal way to behave in the environment in order to carry out his task in the best possible way.



10

10

Key Idea… a bit more formally

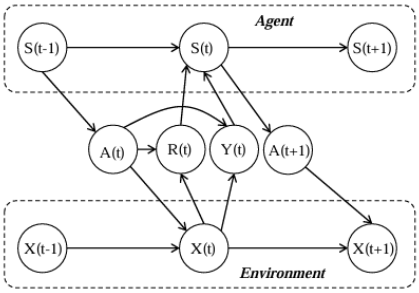


Fig. 3. Model of reinforcement learning problem adapted from [50].

Both the environment and the agent are modeled as stochastic finite state machines. The agent receives observations and rewards as inputs. The outputs from the agent represent actions sent back to the environment. The policy function is $A(t) = \pi(S(t))$ and the state transition function is $S(t) = f(S(t-1), Y(t), R(t), A(t))$. The goal of the agent is to accumulate as much reward as possible. That can be done with a policy and agent state-update function that maximizes the expected value of the summation of rewards:

$$E[R(0) + \gamma R(1) + \gamma^2 R(2) + ...] = E[\sum_{t=0}^{\infty} \gamma^t R(t)] \quad (18)$$

where $0 \leq \gamma \leq 1$ represents the discount factor. This factor signifies that immediate reward is worth more than future reward [20].

11

11

First exposure to some terminology and Several central concepts to RL

12

12

Simulated Environments

Intro to ~~OpenAI Gym~~ Gymnasium

- NOTE: See updated chapter in A.Geron's book:
  - https://github.com/ageron/handson-ml3/blob/main/18_reinforcement_learning.ipynb
- One of the challenges of RL is that in order to train an agent, you first need to have a working environment.
  - If you want to program an agent that will learn to play an Atari game, you will need an Atari game simulator.
- Cannot speed up time either—adding more computing power won't make the robot move any faster—and it's generally too expensive to train 1,000 robots in parallel
- Training is hard and slow in the real world, so you generally need a simulated environment
- **OpenAI Gym** is a toolkit that provides a wide variety of simulated environments (Atari games, board games, 2D and 3D physical simulations, and so on), that you can use to train agents, compare them, or develop new RL algorithms
  - https://github.com/openai/gym

13

Example – CartPole Environment

- Classic control task
- 2D simulation in which a cart can be accelerated left or right in order to balance a pole placed on top of it

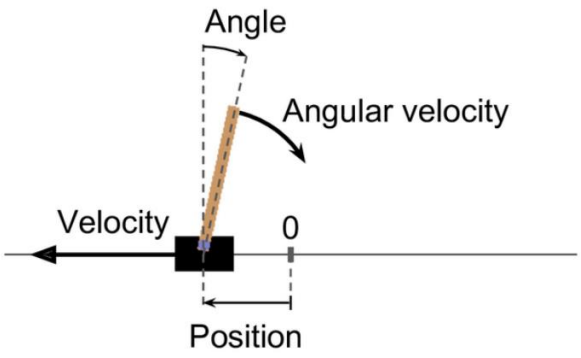


Figure 18-4. The CartPole environment

14

Policy search

A policy, π, determines the probability of choosing each action, a, while the agent is at state s.

The algorithm a software agent uses to determine its actions is called its *policy*. The policy could be a neural network taking observations as inputs and outputting the action to take (see Figure 18-2).

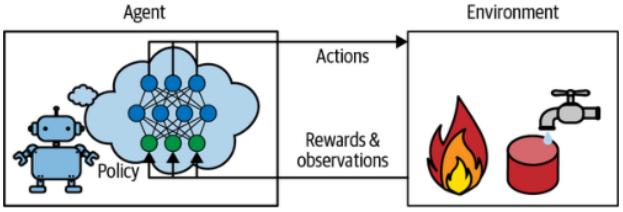


Figure 18-2. Reinforcement learning using a neural network policy

The policy can be any algorithm you can think of, and it does not have to be deterministic. In fact, in some cases it does not even have to observe the environment! For example, consider a robotic vacuum cleaner whose reward is the amount of dust it picks up in 30 minutes. Its policy could be to move forward with some probability p every second, or randomly rotate left or right with probability $1 – p$. The rotation angle would be a random angle between $–r$ and $+r$. Since this policy involves some randomness, it is called a *stochastic policy*. The robot will have an erratic trajectory, which guarantees that it will eventually get to any place it can reach and pick up all the dust. The question is, how much dust will it pick up in 30 minutes?

15

15

Neural Network Policies

- Neural network will take an observation as input, and it will output the action to be executed
- More precisely, it will estimate a probability for each action, then we will **select an action randomly, according to the estimated probabilities**
- For CartPole environment
  - There are just two possible actions (left or right) - only need **one output neuron**
  - It will output the **probability p of action 0 (left),** and the probability of action 1 (right) will be (1 – p)
  - For example, if it outputs 0.7, then we will pick action 0 with 70% probability, or action 1 with 30% probability.

16

16

Exploration/exploitation dilemma is central in RL

- You may wonder why we are picking a random action based on the probabilities given by the neural network, rather than just picking the action with the highest score.
- This approach lets the agent find the right balance between **exploring** new actions and **exploiting** the actions that are known to work well.

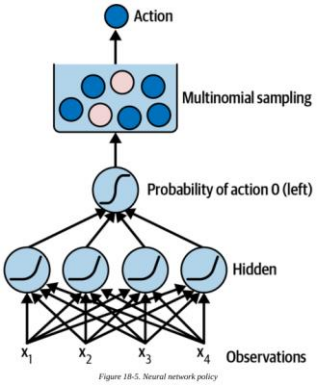


Figure 18-5. Neural network policy

During the learning process, the agent must try different actions (**exploration**) so that it can progressively learn which actions to prefer and perform more often (**exploitation**) in order to maximize the total, cumulative reward. To understand this concept, let's consider a very simple example where a

17

17

Evaluating Actions: the Credit Assignment Problem

If we knew what the best action was at each step, we could train the neural network as usual, by minimizing the cross entropy between the estimated probability distribution and the target probability distribution. It would just be regular supervised learning. However, in reinforcement learning the only guidance the agent gets is through rewards, and rewards are typically sparse and delayed. For example, if the agent manages to balance the pole for 100 steps, how can it know which of the 100 actions it took were good, and which of them were bad? All it knows is that the pole fell after the last action, but surely this last action is not entirely responsible. This is called the *credit assignment problem*: when the agent gets a reward, it is hard for it to know which actions should get credited (or blamed) for it. Think of a dog that gets rewarded hours after it behaved well; will it understand what it is being rewarded for?

unresolved challenges. One aspect that makes training RL models particularly challenging is that the consequent model inputs depend on actions taken previously. This can lead to all sorts of problems, and usually results in unstable learning behavior. Also, this sequence-dependence in RL creates a so-called *delayed effect*, which means that the action taken at a time step t may result in a future reward appearing some arbitrary number of steps later.

18

18

- To tackle this problem, a common strategy is to evaluate an action based on the **sum of all the rewards that come after it**, usually applying a **discount factor, γ (gamma),** at each step.
- This sum of discounted rewards is called the action's **Return.**

Example – Return

action's *return*. Consider the example in Figure 18-6. If an agent decides to go right three times in a row and gets +10 reward after the first step, 0 after the second step, and finally –50 after the third step, then assuming we use a discount factor $\gamma = 0.8$, the first action will have a return of $10 + \gamma \times 0 + \gamma^2 \times (-50) = -22$. If the discount factor is close to 0, then future rewards won't count for much compared to immediate rewards. Conversely, if the discount factor is close to 1, then rewards far into the future will count almost as much as immediate rewards. Typical discount factors vary from 0.9 to 0.99. With a discount factor of 0.95, rewards 13 steps into the future count roughly for half as much as immediate rewards (since $0.95^{13} \approx 0.5$), while with a discount factor of 0.99, rewards 69 steps into the future count for half as much as immediate rewards. In the CartPole environment, actions have fairly short-term effects, so choosing a discount factor of 0.95 seems reasonable.

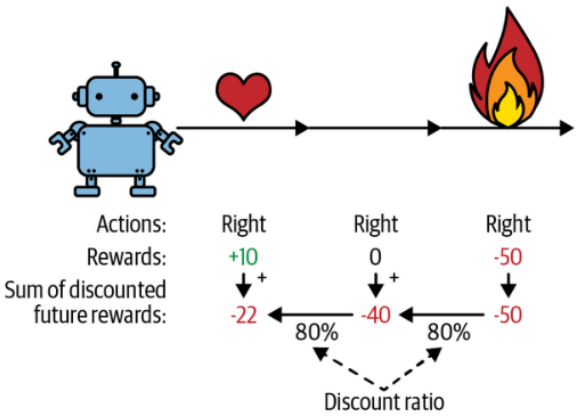
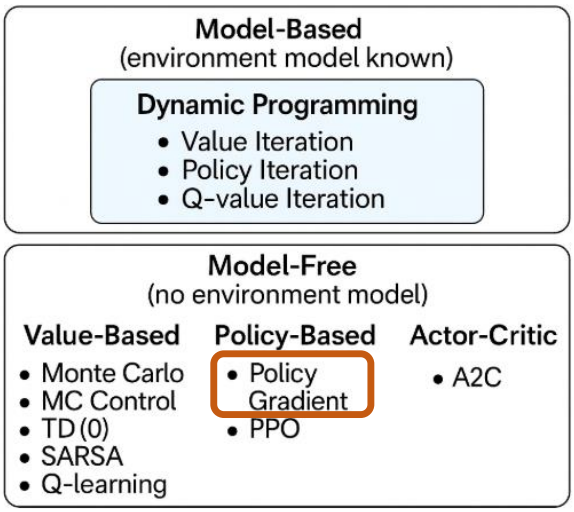


Figure 18-6. Computing an action's return: the sum of discounted future rewards

Big Map of Reinforcement Learning Methods



21

21

Policy Gradients Algorithms

Directly try to optimize the policy to increase rewards

22

22

How to actually evaluate each **action**?

- A good action may be followed by several bad actions that cause the pole to fall quickly, resulting in the good action getting a low return.
- However, if we play the game enough times, **on average** good actions will get a higher return than bad ones.
- **We want to estimate how much better or worse an action is, compared to the other possible actions, on average**.
- This is called the **Action Advantage.**
- For this, we must:
  - Run many **episodes** and
  - Normalize all the action returns, by subtracting the mean and dividing by the standard deviation.
- After that, we can reasonably assume:
  - Actions with a **negative Advantage** were bad while
  - Actions with a **positive Advantage** were good.
- **This strategy is used by (1) Policy Gradients (PG) Algorithms**

23

23

Policy Gradients (PG) Algorithms

As discussed earlier, PG algorithms optimize the parameters of a policy by following the gradients toward higher rewards. One popular class of PG algorithms, called *REINFORCE algorithms*, was introduced back in 1992[11] by Ronald Williams. Here is one common variant:

1. First, let the neural network policy play the game several times, and at each step, compute the gradients that would make the chosen action even more likely—but don't apply these gradients yet.
2. Once you have run several episodes, compute each action's advantage, using the method described in the previous section.
3. If an action's advantage is positive, it means that the action was probably good, and you want to apply the gradients computed earlier to make the action even more likely to be chosen in the future. However, if the action's advantage is negative, it means the action was probably bad, and you want to apply the opposite gradients to make this action slightly *less* likely in the future. The solution is to multiply each gradient vector by the corresponding action's advantage.
4. Finally, compute the mean of all the resulting gradient vectors, and use it to perform a gradient descent step.

24

24

Keras Implementation

We are almost ready to run the algorithm! Now let's define the hyperparameters. We will run 150 training iterations, playing 10 episodes per iteration, and each episode will last at most 200 steps. We will use a discount factor of 0.95:

```
n_iterations = 150
n_episodes_per_update = 10
n_max_steps = 200
discount_factor = 0.95
```

We also need an optimizer and the loss function. A regular Nadam optimizer with learning rate 0.01 will do just fine, and we will use the binary cross-entropy loss function because we are training a binary classifier (there are two possible actions—left or right):

```
optimizer = tf.keras.optimizers.Nadam(learning_rate=0.01)
loss_fn = tf.keras.losses.binary_crossentropy
```

We are now ready to build and run the training loop!

```
for iteration in range(n_iterations):
    all_rewards, all_grads = play_multiple_episodes(
        env, n_episodes_per_update, n_max_steps, model, loss_fn)
    all_final_rewards = discount_and_normalize_rewards(all_rewards,
                                                       discount_factor)
    all_mean_grads = []
    for var_index in range(len(model.trainable_variables)):
        mean_grads = tf.reduce_mean(
            [final_reward * all_grads[episode_index][step][var_index]
             for episode_index, final_rewards in enumerate(all_final_rewards)
                 for step, final_reward in enumerate(final_rewards)], axis=0)
        all_mean_grads.append(mean_grads)

    optimizer.apply_gradients(zip(all_mean_grads, model.trainable_variables))
```

25

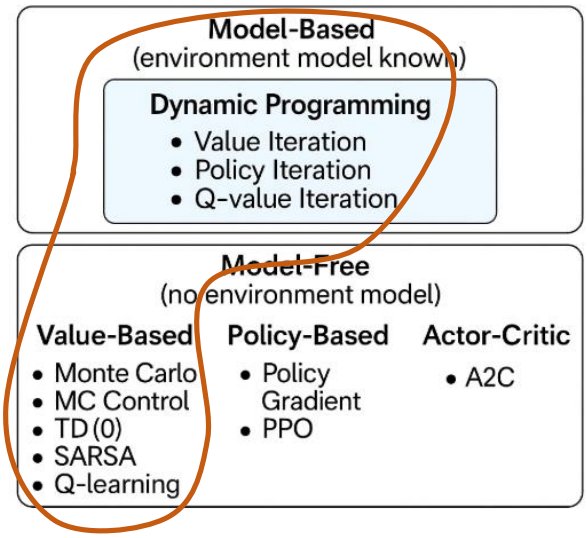
PG limitations

- PG does not scale well to larger and more complex tasks
- It is highly sample inefficient
  - Meaning it needs to explore the game for a very long time before it can make significant progress.
  - This is due to the fact that it must run multiple **episodes** to estimate the advantage of each action.
- However, it is the foundation of more powerful algorithms

26

26

Big Map of Reinforcement Learning Methods



27

27

Different Class of Algorithms

Agent learns to estimate the expected return for each state, or for each action in each state, then it uses this knowledge to decide how to act.

But, first we must talk about:
(A)→MDPs
(B)→More RL terminology
(C)→Bellman equation

28

28

(A)…and before talking about MDPs, we must talk Markov Chains

In the early 20th century, the mathematician Andrey Markov studied stochastic processes with no memory, called *Markov chains*. Such a process has a fixed number of states, and it randomly evolves from one state to another at each step. The probability for it to evolve from a state s to a state s' is fixed, and it depends only on the pair (s, s'), not on past states. This is why we say that the system has no memory.

Figure 18-7 shows an example of a Markov chain with four states.

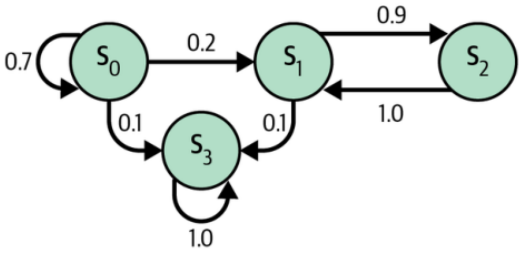


Figure 18-7. Example of a Markov chain

Suppose that the process starts in state s_0, and there is a 70% chance that it will remain in that state at the next step. Eventually it is bound to leave that state and never come back, because no other state points back to s_0. If it goes to state s_1, it will then most likely go to state s_2 (90% probability), then immediately back to state s_1 (with 100% probability). It may alternate a number of times between these two states, but eventually it will fall into state s_3 and remain there forever, since there's no way out: this is called *a terminal state*. Markov chains can have very different dynamics, and they are heavily used in thermodynamics, chemistry, statistics, and much more.

29

29

Markov Decision Processes (MDPs)

Markov decision processes were first described in the 1950s by Richard Bellman.[2] They resemble Markov chains, but with a twist: at each step, an agent can choose one of several possible actions, and the transition probabilities depend on the chosen action. Moreover, some state transitions return some reward (positive or negative), and the agent's goal is to find a policy that will maximize reward over time.

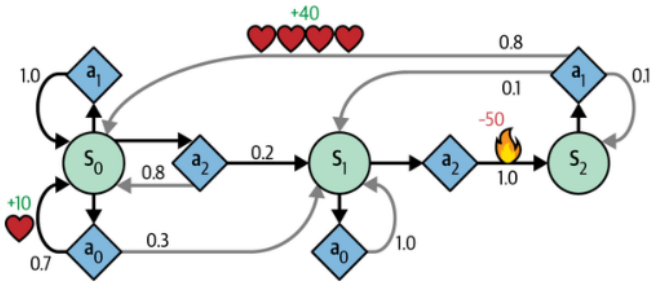


Figure 18-8. Example of a Markov decision process

a_1, which will most likely lead it back to state s_0, gaining a reward of +40 on the way. You get the picture. By looking at this MDP, can you guess which strategy will gain the most reward over time? In state s_0 it is clear that action a_0 is the best option, and in state s_2 the agent has no choice but to take action a_1, but in state s_1 it is not obvious whether the agent should stay put (a_0) or go through the fire (a_2).

30

30

Markov Decision Processes

- Generally, the type of problems that RL deals with are typically formulated as MDPs.
- The standard approach for solving MDP problems is by using dynamic programming, but RL offers some key advantages over dynamic programming.
- Dynamic programming is not a feasible approach when the number of possible configurations is large. In such cases, RL is considered a much more efficient and practical alternative.

Dynamic programming

Dynamic programming refers to a set of computer algorithms and programming methods that was developed by Richard Bellman in the 1950s. In a sense, dynamic programming is about recursive problem solving—solving relatively complicated problems by breaking them down into smaller subproblems.

The key difference between recursion and dynamic programming is that dynamic programming stores the results of subproblems (usually as a dictionary or other form of lookup table) so that they can be accessed in constant time (instead of recalculating them) if they are encountered again in future.

Examples of some famous problems in computer science that are solved by dynamic programming include sequence alignment and computing the shortest path from point A to point B.

MDPs - Mathematical Formulation

$$\{S_0, A_0, R_1\}, \quad \{S_1, A_1, R_2\}, \quad \{S_2, A_2, R_3\}, \quad \ldots$$

that S_t, R_{t+1}, and A_t are time-dependent random variables that take values from predefined finite sets denoted by $s \in \hat{S}$, $r \in \hat{R}$, and $a \in \hat{A}$, respectively. In an MDP, these time-dependent random variables, S_t and R_{t+1}, have probability distributions that only depend on their values at the preceding time step, $t-1$. The probability distribution for $S_{t+1} = s'$ and $R_{t+1} = r$ can be written as a conditional probability over the preceding state (S_t) and taken action (A_t) as follows:

$$p(s', r|s, a) \stackrel{\text{def}}{=} P(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a)$$

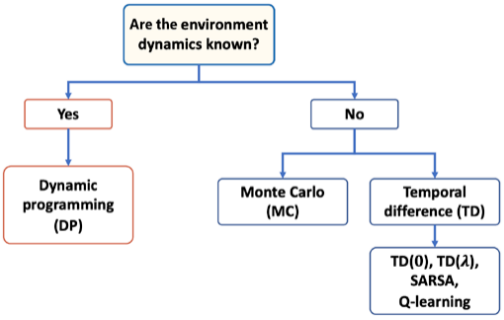
This probability distribution completely defines the **dynamics of the environment** (or model of the environment) because, based on this distribution, all transition probabilities of the environment can be computed. Therefore, the environment dynamics are a central criterion for categorizing different RL methods. The types of RL methods that require a model of the environment or try to learn a model of the environment (that is, the environment dynamics) are called *model-based* methods, as opposed to *model-free* methods.

Model-free and Model-based RL

When the probability $p(s', r|s, a)$ is known, then the learning task can be solved with dynamic programming. But when the dynamics of the environment are not known, as is the case in many real-world problems, then we would need to acquire a large number of samples by interacting with the environment to compensate for the unknown environment dynamics.

Two main approaches for dealing with this problem are:

1. model-free Monte Carlo (MC) and
2. temporal difference (TD) methods



33

33

Generally: environment has **stochastic behavior**

To make sense of this stochastic behavior, let's consider the probability of observing the future state $S_{t+1} = s'$ conditioned on the current state $S_t = s$ and the performed action $A_t = a$. This is denoted by:

$$p(s'|s, a) \stackrel{\text{def}}{=} P(S_{t+1} = s'|S_t = s, A_t = a)$$

It can be computed as a marginal probability by taking the sum over all possible rewards:

$$p(s'|s, a) \stackrel{\text{def}}{=} \sum_{r \in \hat{R}} p(s', r|s, a)$$

This probability is called **state-transition probability**. Based on the state-transition probability, if the environment dynamics are deterministic, then it means that when the agent takes action $A_t = a$ at state $S_t = s$, the transition to the next state, $S_{t+1} = s'$, will be 100 percent certain, that is, $p(s'|s, a) = 1$.

34

34

(B) More RL terminology:

- Return
- Policy
- (State-)value function
- Action-value function

The Return

The so-called *return* at time t is the cumulated reward obtained from the entire duration of an episode. Recall that $R_{t+1} = r$ is the *immediate reward* obtained after performing an action, A_t, at time t; the *subsequent* rewards are R_{t+2}, R_{t+3}, and so forth.

The return at time t can then be calculated from the immediate reward as well as the subsequent ones, as follows:

$$G_t \stackrel{\text{def}}{=} R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0} \gamma^k R_{t+k+1}$$

$$G_t = R_{t+1} + \gamma G_{t+1} = r + \gamma G_{t+1}$$

Here, γ is the *discount factor* in range [0, 1]. The parameter γ indicates how much the future rewards are "worth" at the current moment (time t). Note that by setting $\gamma = 0$, we would imply that we do not care about future rewards. In this case, the return will be equal to the immediate reward, ignoring the subsequent rewards after $t + 1$, and the agent will be short-sighted. On the other hand, if $\gamma = 1$, the return will be the unweighted sum of all subsequent rewards.

Policy

A *policy* typically denoted by $\pi(a|s)$ is a function that determines the next action to take, which can be either deterministic or stochastic (that is, the probability for taking the next action). A stochastic policy then has a probability distribution over actions that an agent can take at a given state:

$$\pi(a|s) \stackrel{\text{def}}{=} P[A_t = a|S_t = s]$$

During the learning process, the policy may change as the agent gains more experience. For example, the agent may start from a random policy, where the probability of all actions is uniform; meanwhile, the agent will hopefully learn to optimize its policy toward reaching the optimal policy. The *optimal policy* $\pi_*(a|s)$ is the policy that yields the highest return.

37

37

State-value function (or value function)

The *value function*, also referred to as the *state-value function*, measures the *goodness* of each state—in other words, how good or bad it is to be in a particular state. Note that the criterion for goodness is based on the return.

Now, based on the return G_t, we define the value function of state s as the expected return (the average return over all possible episodes) after *following policy* π:

$$v_\pi(s) \stackrel{\text{def}}{=} E_\pi[G_t|S_t = s] = E_\pi\left[\sum_{k=0} \gamma^{k+1} R_{t+k+1} \middle| S_t = s\right]$$

In an actual implementation, we usually estimate the value function using lookup tables, so we do not have to recompute it multiple times. (This is the dynamic programming aspect.) For example,

38

38

Action-value function

Moreover, we can also define a value for each state-action pair, which is called the *action-value function* and is denoted by $q_\pi(s, a)$. The action-value function refers to the expected return G_t when the agent is at state $S_t = s$ and takes action $A_t = a$.

Extending the definition of the state-value function to state-action pairs, we get the following:

$$q_\pi(s, a) \stackrel{\text{def}}{=} E_\pi[G_t | S_t = s, A_t = a] = E_\pi\left[\sum_{k=0} \gamma^{k+1} R_{t+k+1} \middle| S_t = s, A_t = a\right]$$

This is similar to referring to the optimal policy as $\pi_*(a|s)$, $v_*(s)$, and $q_*(s, a)$ also denote the optimal state-value and action-value functions.

Estimating the value function is an essential component of RL methods.
There are different ways of calculating and estimating the (state-)value function and action-value function.

39

39

(C) The Belman Equation

- The Bellman equation is one of the central elements of many RL algorithms.
- The Bellman equation simplifies the computation of the value function, such that rather than summing over multiple time steps, it uses a recursion that is similar to the recursion for computing the return.

Now, we can see that expectation of the return, $E_\pi[G_{t+1}|S_t = s']$, is essentially the state-value function $v_\pi(s')$. So, we can write $v_\pi(s)$ as a function of $v_\pi(s')$:

$$v_\pi(s) = \sum_{a \in \hat{A}} \pi(a|s) \sum_{s' \in \hat{S}, r \in \hat{R}} p(s', r|s, a)[r + \gamma\, v_\pi(s')]$$

This is called the **Bellman equation**, which relates the value function for a state, s, to the value function of its subsequent state, s'. This greatly simplifies the computation of the value function because it eliminates the iterative loop along the time axis.

40

40

RL Algorithms

Figure 19.5 describes the course of advancing RL algorithms, from dynamic programming to Q-learning:

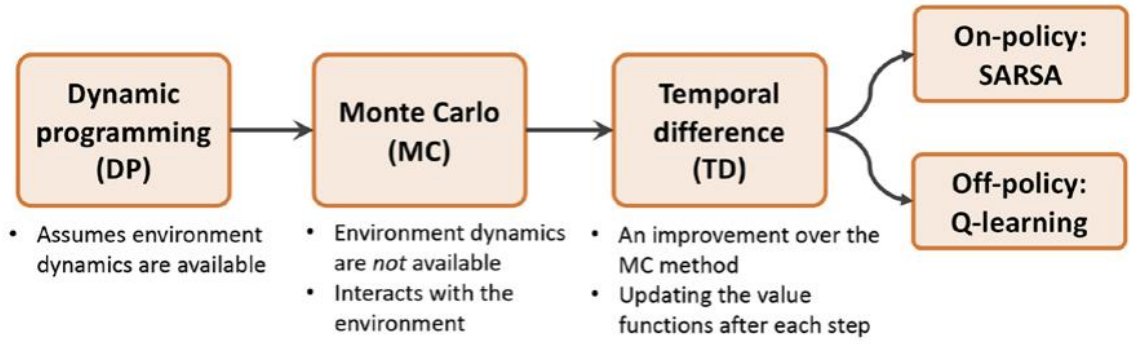


Figure 19.5: Different types of RL algorithms

41

41

1.Dynamic Programming (DP)

- Problem with using dynamic programming is that it assumes full knowledge of the environment dynamics.
- Because of that it is not very practical; but helpful to introduce more advanced/complicated RL algorithms.
- **Assumptions:**
  - We have full knowledge of the environment dynamics; that is, all transition probabilities $p(s', r|s, a)$—are known.
  - The agent's state has the Markov property, which means that the next action and reward depend only on the current state and the choice of action we make at this moment or current time step.
- **Objectives:**
  1. Obtain the true state-value function, $v_\pi(s)$; this task is also known as the prediction task and is accomplished with *policy evaluation*.
  2. Find the optimal value function, $v_*(s)$, which is accomplished via *generalized policy iteration*.

42

(i)Policy evaluation – predicting the value function with DP (This is the prediction task)

Based on the Bellman equation, we can compute the value function for an arbitrary policy π with dynamic programming when the environment dynamics are known. For computing this value function, we can adapt an iterative solution, where we start from $v^{\langle 0 \rangle}(s)$, which is initialized to zero values for each state. Then, at each iteration $i+1$, we update the values for each state based on the Bellman equation, which, in turn, is based on the values of states from a previous iteration, i, as follows:

$$v^{\langle i+1 \rangle}(s) = \sum_{a} \pi(a|s) \sum_{s' \in \hat{S}, r \in \hat{R}} p(s', r|s, a)\left[r + \gamma\, v^{\langle i \rangle}(s')\right]$$

It can be shown that as the iterations increase to infinity, $v^{\langle i \rangle}(s)$ converges to the true state-value function, $v_\pi(s)$.

- Notice here that we do not need to interact with the environment.
- **Computed value function can be used to improve the arbitrary policy!**

43

43

(ii) Improving the policy - using the estimated value function (This is the control task)

Now that we have computed the value function $v_\pi(s)$ by following the existing policy, π, we want to use $v_\pi(s)$ and improve the existing policy, π. This means that we want to find a new policy, π', that, for each state, s, following π', would yield higher or at least equal value than using the current policy, π. In mathematical terms, we can express this objective for the improved policy, π', as:

$$v_{\pi'}(s) \geq v_\pi(s) \quad \forall s \in \hat{S}$$

The policy improvement algorithm:

- First, compute the action-value function, $q_\pi(s,a)$, for each state, s, and action, a, based on the computed state value using value function $v_\pi(s)$
- Iterate through all the states, and for each state, s, compare the value of the next state, s', that would occur if action a was selected
- Compare the corresponding action with the action selected by the current policy
- If action suggested by current policy (i.e., *arg max* $\pi(a|s)$) is different than action suggested by action-value function (i.e., *arg max* $q_\pi(s,a)$), then update policy by reassigning probabilities of actions to match action that gives highest action value, $q_\pi(s,a)$

44

44

- It can be shown that the policy improvement will strictly yield a better policy, unless the current policy is already optimal
- This technique is referred to as **generalized policy iteration (GPI)**, which is common among many RL methods.
  - GPI used for the MC and TD learning methods.
- For efficiency reasons: **Combine the two tasks of policy evaluation and policy improvement into a single step:**

The following equation updates the value function for iteration $i + 1$ (denoted by $v^{(i+1)}$) based on the action that maximizes the weighted sum of the next state value and its immediate reward $(r + \gamma\, v^{(i)}(s'))$:

$$v^{(i+1)}(s) = \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma\, v^{(i)}(s')\right]$$

In this case, the updated value for $v^{(i+1)}(s)$ is maximized by choosing the best action out of all possible actions, whereas in policy evaluation, the updated value was using the weighted sum over all actions.

NOTE: Read the section on the optimal State Value (pp. 701, equation 18.1) in A.Geron's book.

45

45

Value iteration Algorithm

This equation leads directly to an algorithm that can precisely estimate the optimal state value of every possible state: first initialize all the state value estimates to zero, and then iteratively update them using the *value iteration* algorithm (see Equation 18-2). A remarkable result is that, given enough time, these estimates are guaranteed to converge to the optimal state values, corresponding to the optimal policy.

Equation 18-2. Value iteration algorithm

$$V_{k+1}(s) \leftarrow \max_a \sum_{s\prime} T(s, a, s\prime)\left[R(s, a, s\prime) + \gamma \cdot V_k(s\prime)\right] \quad \text{for all } s$$

In this equation, $V_k(s)$ is the estimated value of state s at the k^{th} iteration of the algorithm.

46

46

Q-value iteration Algorithm

Knowing the optimal state values can be useful, in particular to evaluate a policy, but it does not give us the optimal policy for the agent. Luckily, Bellman found a very similar algorithm to estimate the optimal *state-action values*, generally called *Q-values* (quality values). The optimal Q-value of the state-action pair (*s*, *a*), noted $Q^*(s, a)$, is the sum of discounted future rewards the agent can expect on average after it reaches the state *s* and chooses action *a*, but before it sees the outcome of this action, assuming it acts optimally after that action.

Let's look at how it works. Once again, you start by initializing all the Q-value estimates to zero, then you update them using the *Q-value iteration* algorithm (see Equation 18-3).

Equation 18-3. Q-value iteration algorithm

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \cdot \max_{a'} Q_k(s',a') \right] \quad \text{for all } (s,a)$$

Once you have the optimal Q-values defining the optimal policy, noted $\pi^*(s)$, is trivial; when the agent is in state *s*, it should choose the action with the highest Q-value for that state: $\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q^*(s,a)$.

NOTE: See this algorithm applied to an example MDP in Geron's book.

47

47

A Note on Notations

Notation for tabular estimates of the state-value and action-value functions

In most RL literature and textbooks, the lowercase v_π and q_π are used to refer to the true state-value and true action-value functions, respectively, as mathematical functions.

Meanwhile, for practical implementations, these value functions are defined as lookup tables. The tabular estimates of these value functions are denoted by $V(S_t = s) \approx v_\pi(s)$ and $Q_\pi(S_t = s, A_t = a) \approx q_\pi(s,a)$. We will also use this notation in this chapter.

48

48

2.Monte Carlo

- Assume we do not have any knowledge about the environment dynamics
  - **We do not know the state-transition probabilities of the environment**
- Instead, we want the agent to learn through **interacting** with the environment.
- Using MC methods, the learning process is based on the so-called **simulated experience**
- MC-based RL methods:
  - Define an agent class that follows a probabilistic policy, π, and based on this policy, agent takes an action at each step. This results in a simulated episode.
  - Generate simulated episodes.
  - From these simulated episodes, we compute the average return for each state visited (during an episode).

(i) State-value function estimation using MC

After generating a set of episodes, for each state, s, the set of episodes that all pass through state s is considered for calculating the value of state s. Let's assume that a lookup table is used for obtaining the value corresponding to the value function, $V(S_t = s)$. MC updates for estimating the value function are based on the total return obtained in that episode starting from the first time that state s is visited. This algorithm is called *first-visit Monte Carlo* value prediction.

Let's first revisit the value prediction by MC. At the end of each episode, we are able to estimate the return, G_t, for each time step t. Therefore, we can update our estimates for the visited states as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$$

Here, G_t is used as the *target return* to update the estimated values, and $(G_t - V(S_t))$ is a *correction* term added to our current estimate of the value $V(S_t)$. The value α is a hyperparameter denoting the learning rate, which is kept constant during learning.

Action-value function estimation using MC

- Extend the algorithm for estimating the first-visit MC state-value prediction.
- Compute estimated return for each state-action pair using the action-value function: consider visits to each state-action pair (s, a).
- A problem arises: some actions may never be selected → insufficient exploration.
- Ways to resolve this:
  1. Exploratory start - assumes every state-action pair has a non-zero probability at the beginning of the episode.
  2. The ϵ-greedy policy

(ii) Finding an **optimal policy** using MC control

MC control refers to the optimization procedure for improving a policy. Similar to the policy iteration approach in the previous section (*Dynamic programming*), we can repeatedly alternate between policy evaluation and policy improvement until we reach the optimal policy. So, starting from a random policy, π_0, the process of alternating between policy evaluation and policy improvement can be illustrated as follows:

$$\pi_0 \xrightarrow{Eval.} q_{\pi_0} \xrightarrow{Improve} \pi_1 \xrightarrow{Eval.} q_{\pi_1} \xrightarrow{Improve} \pi_2 \quad \ldots \quad \xrightarrow{Eval.} q_* \xrightarrow{Improve} \pi_*$$

3.Temporal Difference (TD) learning

Similar to the MC technique, TD learning is also based on learning by experience and, therefore, does not require any knowledge of environment dynamics and transition probabilities. The main difference between the TD and MC techniques is that in MC, we have to wait until the end of the episode to be able to calculate the total return.

However, in TD learning, we can leverage some of the learned properties to update the estimated values before reaching the end of the episode. This is called *bootstrapping* (in the context of RL, the term *bootstrapping* is not to be confused with the bootstrap estimates we used in *Chapter 7*, *Combining Different Models for Ensemble Learning*).

TD learning algorithm

Reinforcement learning problems with discrete actions can often be modeled as Markov decision processes, but the agent initially has no idea what the transition probabilities are (it does not know $T(s, a, s')$), and it does not know what the rewards are going to be either (it does not know $R(s, a, s')$). It must experience each state and each transition at least once to know the rewards, and it must experience them multiple times if it is to have a reasonable estimate of the transition probabilities.

The *temporal difference (TD) learning* algorithm is very similar to the Q-value iteration algorithm, but tweaked to take into account the fact that the agent has only partial knowledge of the MDP. In general we assume that the agent initially knows only the possible states and actions, and nothing more. The agent uses an *exploration policy* —for example, a purely random policy—to explore the MDP, and as it progresses, the TD learning algorithm updates the estimates of the state values based on the transitions and rewards that are actually observed (see Equation 18-4).

Equation 18-4. TD Learning algorithm

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$$

or, equivalently:

$$V_{k+1}(s) \leftarrow V_k(s) + \alpha \cdot \delta_k(s, r, s')$$

with $\delta_k(s, r, s') = r + \gamma \cdot V_k(s') - V_k(s)$

In this equation:

- α is the learning rate (e.g., 0.01).
- $r + \gamma \cdot V_k(s')$ is called the *TD target*.
- $\delta_k(s, r, s')$ is called the *TD error*.

(i) Value function prediction

In TD learning, we replace the actual return, $G_{t:T}$, with a new target return, $G_{t:t+1}$, which significantly simplifies the updates for the value function, $V(S_t)$. The update formula based on TD learning is as follows:

Return obtained at time step t --> t+1

$$V(S_t) \leftarrow V(S_t) + \alpha(G_{t:t+1} - V(S_t))$$

Here, the target return, $G_{t:t+1} \stackrel{\text{def}}{=} R_{t+1} + \gamma V(S_{t+1}) = r + \gamma V(S_{t+1})$, is using the observed reward, R_{t+1}, and the estimated value of the next immediate step. Notice the difference between MC and TD. In MC, $G_{t:T}$ is not available until the end of the episode, so we should execute as many steps as needed to get there. On the contrary, in TD, we only need to go one step ahead to get the target return. This is also known as TD(0).

TD(0) algorithm can be generalized to the so-called n-step TD algorithm

MC versus TD: which method converges faster?

While the precise answer to this question is still unknown, in practice, it is empirically shown that TD can converge faster than MC. If you are interested, you can find more details on the convergences of MC and TD in the book entitled *Reinforcement Learning: An Introduction*, by *Richard S. Sutton* and *Andrew G. Barto*.

55

55

(ii) Two algorithms for TD control:

a) An on-policy control – value function is updated based on the actions from the same policy that the agent is following

b) An off-policy control – value function is updated based on actions outside the current policy

- Like in the case of dynamic programming and MC algorithms, use the **generalized policy iteration (GPI)** for policy improvement

56

56

(ii.a) On-policy TD control (SARSA)

a lookup table, that is, a tabular 2D array, $Q(S_t, A_t)$, which represents the action-value function for each state-action pair. In this case, we will have the following:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma\, Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

This algorithm is often called SARSA, referring to the quintuple (S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}) that is used in the update formula.

As we saw in the previous sections describing the dynamic programming and MC algorithms, we can use the GPI framework, and starting from the random policy, we can repeatedly estimate the action-value function for the current policy and then optimize the policy using the ϵ-greedy policy based on the current action-value function.

57

57

(ii.b) Off-policy TD control (Q-learning)

Instead of updating the action-value function using the action value of A_{t+1} that is taken by the agent, we can find the best action even if it is not actually chosen by the agent following the current policy. (That's why this is considered an *off-policy* algorithm.)

To do this, we can modify the update rule to consider the maximum Q-value by varying different actions in the next immediate state. The modified equation for updating the Q-values is as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

We encourage you to compare the update rule here with that of the SARSA algorithm. As you can see, we find the best action in the next state, S_{t+1}, and use that in the correction term to update our estimate of $Q(S_t, A_t)$.

58

58

Q-learning algorithm

Similarly, the Q-learning algorithm is an adaptation of the Q-value iteration algorithm to the situation where the transition probabilities and the rewards are initially unknown (see Equation 18-5). Q-learning works by watching an agent play (e.g., randomly) and gradually improving its estimates of the Q-values. Once it has accurate Q-value estimates (or close enough), then the optimal policy is just choosing the action that has the highest Q-value (i.e., the greedy policy).

Equation 18-5. Q-learning algorithm

$$Q(s,a) \underset{\alpha}{\leftarrow} r + \gamma \cdot \max_{a'} Q(s',a')$$

For each state-action pair (s, a), this algorithm keeps track of a running average of the rewards r the agent gets upon leaving the state s with action a, plus the sum of discounted future rewards it expects to get. To estimate this sum, we take the maximum of the Q-value estimates for the next state s', since we assume that the target policy will act optimally from then on.

59

59

Obviously, not knowing the transition probabilities or the rewards makes finding the optimal policy significantly harder!

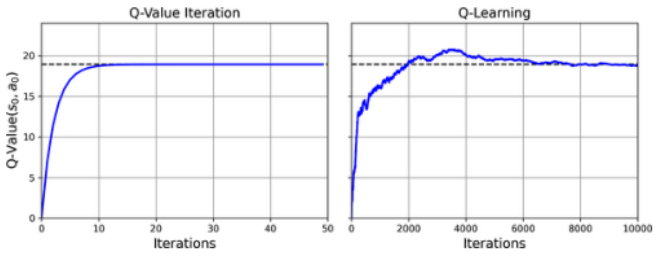


Figure 18-9. Learning curve of the Q-value iteration algorithm versus the Q-learning algorithm

- **Q-learning algorithm** is called an **off-policy algorithm** because the policy being trained is not necessarily the one used during training. After training, the optimal policy corresponds to systematically choosing the action with the highest Q-value.
- In contrast, **Policy Gradients (PG) algorithm** is an **on-policy algorithm**: it explores the world using the policy being trained.

60

60

Exploration Policies

Of course, Q-learning can work only if the exploration policy explores the MDP thoroughly enough. Although a purely random policy is guaranteed to eventually visit every state and every transition many times, it may take an extremely long time to do so. Therefore, a better option is to use the *ε-greedy policy* (ε is epsilon): at each step it acts randomly with probability ε, or greedily with probability $1–\varepsilon$ (i.e., choosing the action with the highest Q-value). The advantage of the ε-greedy policy (compared to a completely random policy) is that it will spend more and more time exploring the interesting parts of the environment, as the Q-value estimates get better and better, while still spending some time visiting unknown regions of the MDP. It is quite common to start with a high value for ε (e.g., 1.0) and then gradually reduce it (e.g., down to 0.05).

Alternatively, rather than relying only on chance for exploration, another approach is to encourage the exploration policy to try actions that it has not tried much before. This can be implemented as a bonus added to the Q-value estimates, as shown in Equation 18-6.

Equation 18-6. Q-learning using an exploration function

$$Q\left(s,a\right) \underset{\alpha}{\leftarrow} r + \gamma \cdot \max_{a\prime} f(Q(s\prime,a\prime), N(s\prime,a\prime))$$

In this equation:

- $N(s', a')$ counts the number of times the action a' was chosen in state s'.
- $f(Q, N)$ is an *exploration function*, such as $f(Q, N) = Q + \kappa/(1 + N)$, where κ is a curiosity hyperparameter that measures how much the agent is attracted to the unknown.

61

61

Approximate Q-learning and Deep Q-learning

62

62

Approximate Q-Learning and **Deep Q-Learning**

- The main problem with Q-learning is that it does not scale well to large (or even medium) MDPs with many states and actions.

The solution is to find a function $Q_{\boldsymbol{\theta}}(s, a)$ that approximates the Q-value of any state-action pair (s, a) using a manageable number of parameters (given by the parameter vector $\boldsymbol{\theta}$). This is called *approximate Q-learning*. For years it was recommended to use linear combinations of handcrafted features extracted from the state (e.g., the distances of the closest ghosts, their directions, and so on) to estimate Q-values, but in 2013, DeepMind showed that using deep neural networks can work much better, especially for complex problems, and it does not require any feature engineering. A DNN used to estimate Q-values is called a *deep Q-network* (DQN), and using a DQN for approximate Q-learning is called *deep Q-learning*.

How can we train a DQN?

Now, how can we train a DQN? Well, consider the approximate Q-value computed by the DQN for a given state-action pair (s, a). Thanks to Bellman, we know we want this approximate Q-value to be as close as possible to the reward r that we actually observe after playing action a in state s, plus the discounted value of playing optimally from then on. To estimate this sum of future discounted rewards, we can just execute the DQN on the next state s', for all possible actions a'. We get an approximate future Q-value for each possible action. We then pick the highest (since we assume we will be playing optimally) and discount it, and this gives us an estimate of the sum of future discounted rewards. By summing the reward r and the future discounted value estimate, we get a target Q-value $y(s, a)$ for the state-action pair (s, a), as shown in Equation 18-7.

Equation 18-7. Target Q-value

$$y\left(s,a\right) = r + \gamma \cdot \max_{a\prime} Q_{\boldsymbol{\theta}}\left(s\prime, a\prime\right)$$

With this target Q-value, we can run a training step using any gradient descent algorithm. Specifically, we generally try to minimize the squared error between the estimated Q-value $Q_{\boldsymbol{\theta}}(s, a)$ and the target Q-value $y(s, a)$, or the Huber loss to reduce the algorithm's sensitivity to large errors. And that's the deep Q-learning algorithm!

NOTE: See this algorithm implemented for the CartPole environment in the code of Geron's book.

PART 2

Code Time!

65

65

Code Time

- See demonstration and discussion in class.
- See also links in the “Code Examples” for this lecture assignment.

66

66

Conclusion
Takeaways

67

Reinforcement Learning with a known vs. unknown model

Category	Uses environment model?	Example methods
Dynamic Programming (DP)	YES (transition probabilities & rewards known)	Value Iteration, Policy Iteration, **Q-value Iteration**
Monte Carlo (MC)	NO (learn from full episodes)	Monte Carlo Prediction, Monte Carlo Control
Temporal Difference (TD)	NO (learn from bootstrapped estimates without a model)	TD(0), SARSA, Q-learning

68

References and Credits

Many of the teaching materials for this course have been adapted from various sources. We are very grateful and thank the following professors, researchers, and practitioners for sharing their teaching materials (in no particular order):

- Yaser S. Abu-Mostafa, Malik Magdon-Ismail and Hsuan-Tien Lin. https://amlbook.com/slides.html
- Ethem Alpaydin. https://www.cmpe.boun.edu.tr/~ethem/i2ml3e/
- Natasha Jaques. https://courses.cs.washington.edu/courses/cse446/25sp/
- Lyle Ungar. https://alliance.seas.upenn.edu/~cis520/dynamic/2022/wiki/index.php?n=Lectures.Lectures
- Aurelien Geron. https://github.com/ageron/handson-ml3
- Sebastian Raschka. https://github.com/rasbt/machine-learning-book
- Trevor Hastie. https://www.statlearning.com/resources-python
- Andrew Ng. https://www.youtube.com/playlist?list=PLoROMvodv4rMiGQp3WXShtMGgzqpfVfbU
- Richard Povineli. https://www.richard.povinelli.org/teaching
- … and many others.

69