Reinforcement Learning - Lecture Notes

Sayantan Auddy

November 13, 2020

Contents

These notes are based on the RL course notes from the previous year [https://iis.uibk.ac.at/uibk/piater/courses/IIS/modules/RL/RL.notes.pdf] and also follow the content in Sutton and Barto (2nd edition) [10], as well as the lecture notes of David Silver [http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching.html]. The notation followed is from [10].

1 Introduction

Reinforcement Learning (RL) is an area of machine learning in which the objective is to train an artificial **agent** to perform a given task in a stochastic **environment** by letting it interact with its environment repeatedly (by taking **actions** which affect the environment). While the agent aims to learn how to map observations (**states**) to actions, there is no teacher which provides the correct actions during training. Instead, a scalar feedback signal, known as a **reward**, is provided to the agent by the environment. The agent tries to map observations to actions in a way so that the cumulative reward it collects is maximized.

1.1 A Motivating Example

How would you train a program to play the game of tic-tac-toe? For a simple game such as this, an exhaustive search is possible. Each possible board configuration represents a distinct state of the game. Since the number of possible states is relatively small and manageable, we can maintain a tree-like structure in which the possible moves are maintained (start with the start state as the root node, the possible states reachable from it are the root node's children and so on). The leaf-nodes of this tree would be the terminal states (win or loss configurations). An exhaustive search can then be performed using techniques such as Minimax search [1] to find the game moves that lead to a winning final state. However, such an approach is not scalable. As the number of states and possible actions increases, it becomes impossible to maintain a game tree and to perform an exhaustive search on it.

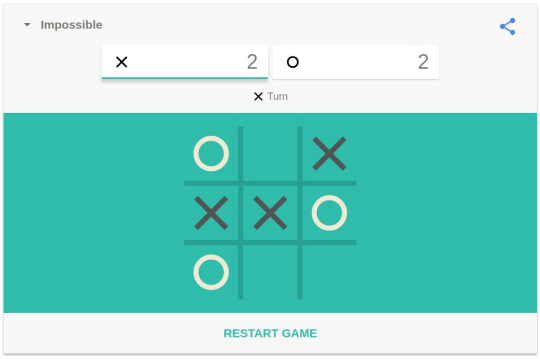


Figure 1: The game of tic-tac-toe [https://www.google.com/search?q=tic+tac+toe]

1.2 A Simplified View of RL

A simple way to visualize the agent-environment interaction is depicted in Fig. 2, where an artificial agent learns to play a video game. Here the environment is comprised of the game console (which controls the rules and dynamics of the game, but the agent cannot access this internal information). At each step, the agent observes the current state of the environment (as seen on the console screen), and based on this observation performs an action by using the controller/joystick. In return it receives a numerical reward in the form of the game score. In this scenario, the objective of an RL algorithm will be to help the agent to learn to play the game by maximizing the game score that it attains.

[1] https://en.wikipedia.org/wiki/Minimax

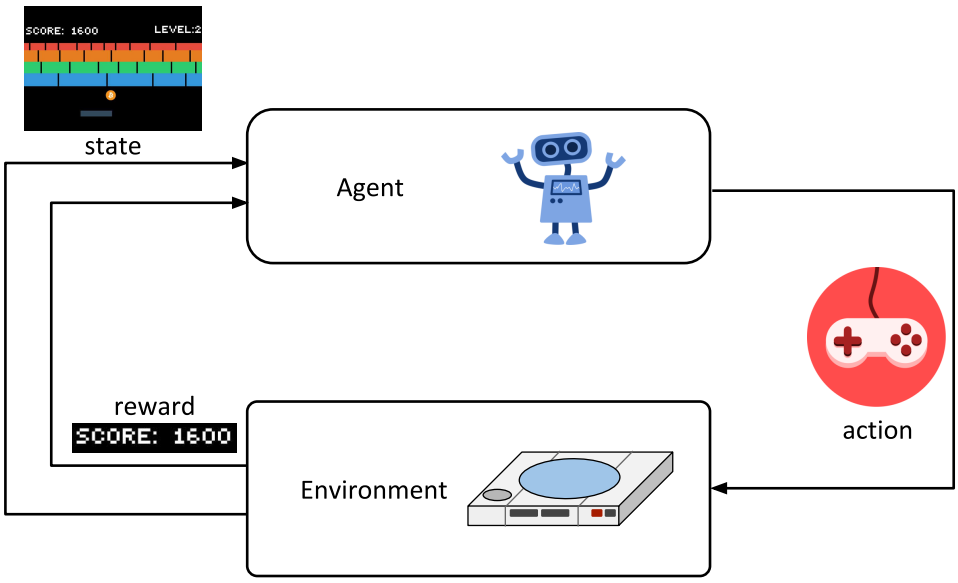


Figure 2: The reinforcement learning loop

For the agent, the reward signal forms a direct sensorimotor connection to the environment - the agent is able to understand the consequences of its actions through the rewards that it receives. However, the consequence of taking a particular action in a given situation may not be apparent immediately, since a good action may lead to a good reward much later after the action has been taken (**delayed reward**). For example, pressing the correct button at time $t = 0$ can be rewarded at time $t = 10$. The agent should also learn to choose actions that lead to good results in the long run rather than actions which give good immediate rewards but are detrimental in the long run. Once the agent discovers a good action for a given observation (state), it can choose to perform the same action whenever this state is encountered (**exploitation**). However, a better ploy would be to sometimes also explore other alternative actions, in case one of the other actions turns out to be even better (**exploration**).

Distinguishing features of Reinforcement Learning

- No explicit teacher
- Learning by trial and error
- Learning through repeated agent-environment interactions
- Goal-oriented learning by maximizing cumulative reward
- Delayed rewards
- Need to balance exploitation vs exploration

1.3 Origins or Reinforcement Learning

The origins of reinforcement learning can be traced to various fields. Most prominent among them are the fields of **psychology** and **control theory**. The concept of learning

through trial and error originated in studies in psychology dealing with animal learning behavior. The concept of optimal control and its solution through the use of value functions and dynamic programming comes from the field of control theory [2]. Studies in computational **neuroscience** focus on developing computational models that can explain the observed behavior in animals. Several of the concepts in reinforcement learning have parallels in neuroscience, for example, the use of a reward signal can be compared with the neurotransmitter called dopamine which mediates pleasure in the brain [3]. In the early years of **artificial intelligence**, trial-and-error learning was explored as an engineering principle [2]. Artificial neural networks have also often been used as a function approximator for implementing reinforcement learning algorithms and have led to many recent advances in the field.

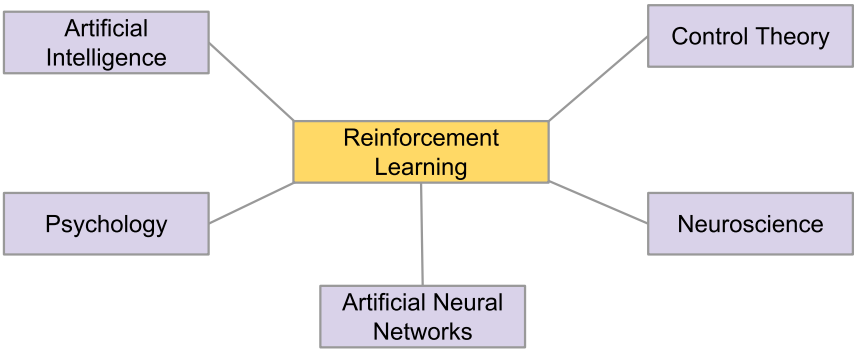


Figure 3: Origins of Reinforcement Learning

1.4 Machine Learning Paradigms

Reinforcement Learning is one of the three different kinds of machine learning techniques. Fig. 4 highlights the key differences between the different machine learning paradigms.

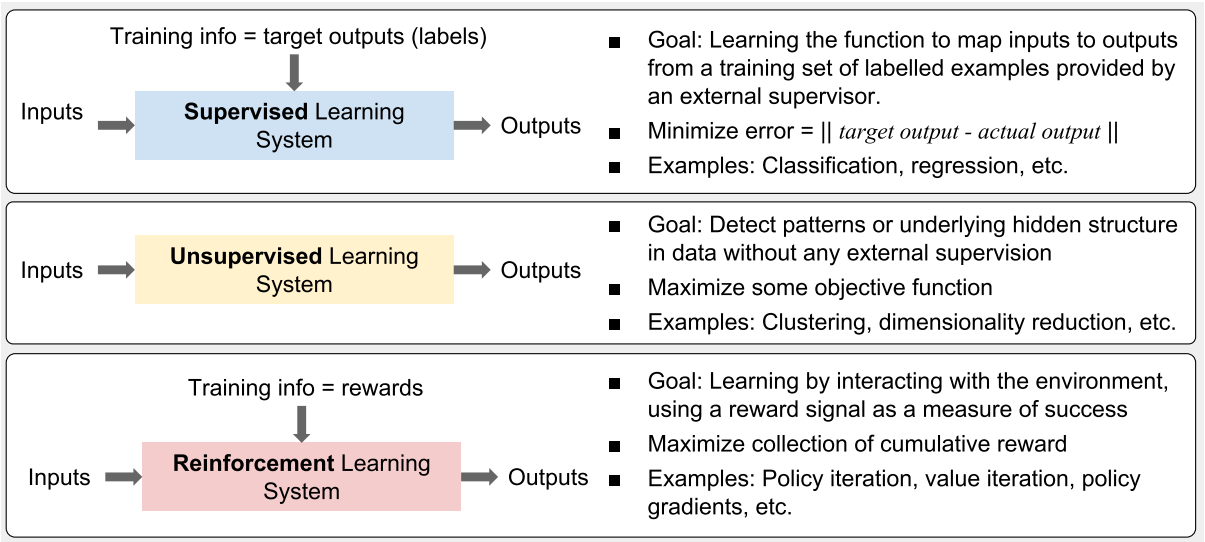


Figure 4: Machine learning paradigms

[2] http://www.incompleteideas.net/book/ebook/node12.html
[3] http://www.princeton.edu/~yael/ICMLTutorial.pdf

1.5 Elements of Reinforcement Learning

Before diving into the formal description of the reinforcement learning setup, let us understand the meaning of some commonly used terms, as listed in Table 1.

Table 1: Informal description of common RL terms.

Term	Description
Agent	The artificial entity that is being trained to perform a task by learning from its own experience. A learning agent must be able to sense the state of its environment and must be able to take actions to affect the state.
Environment	Comprises of everything outside the purview of the agent. The environment has its own internal dynamics and rules which are usually not visible to the agent. The boundary between agent and environment is typically not the same as the physical boundary of a robot's or animal's body. Usually, the boundary is drawn closer to the agent than that.
State	Refers to the current situation of the environment (as observed by the agent), which forms the basis for the decisions taken by the agent.
Action	Choices made by the agent to change the state of the environment.
Reward	The reward signal is a scalar quantity that is emitted by the environment in response to the action taken by the agent. It defines the goal of the reinforcement learning problem and defines what is good in the short term. It forms the basis for evaluating the choices made by the agent.
Return	The cumulative sum of rewards received or expected to be received by the agent.
Goal	The agent's goal is to maximize the total amount of reward (the expected return) it receives.
Policy	Defines the learning agent's behavior. The policy can be viewed as a function that provides a mapping from perceived states to actions (or probability of actions) to be taken in those states. It can be implemented as a lookup table or as a function approximator such as a neural network.
Value Function	Specifies what is good in the long run. Value of a state is the total amount of reward that an agent can expect to accumulate starting from that state. It is used to decide which actions should be taken. The most important component of RL algorithms is a method for efficiently estimating values.
Model	Something that mimics the behavior of the environment and allows inferences to be made about how the environment will behave. In RL, models are used for planning by deciding on a course of action by considering situations even before they have occurred in reality.

1.6 Categories of Reinforcement Learning Algorithms

Based on the type of formulation used, reinforcement learning agents can be categorized as follows:

- **Value-based**: A value function is maintained for estimating the desirability of states and actions. There is no explicit policy function for mapping states to actions.

Actions which result in moving to a state with high value are preferred (implicit policy).

- **Policy-based**: An explicit policy function is learned for mapping states to actions directly, without going through value functions.
- **Actor-Critic**: Both a value function and a policy function are maintained. The value function is utilized for improving the policy.
- **Model-free**: Value and/or policy functions are used without creating a model of the environment.
- **Model-based**: A model of the environment is maintained for predicting the behavior of the environments, in addition to value or policy functions. If an accurate model can be created, then interaction with the environment can be simulated for training the agent.
- ***Tabular Methods***: These methods use tables to list the values of each state or state-action pair. These can only be used if the number of discrete states and actions are not too high.
- ***Approximate Methods***: These methods use some kind of function approximation to represent value and policy functions. They are applicable in cases where the number of states are very high or while dealing with continuous states and actions.

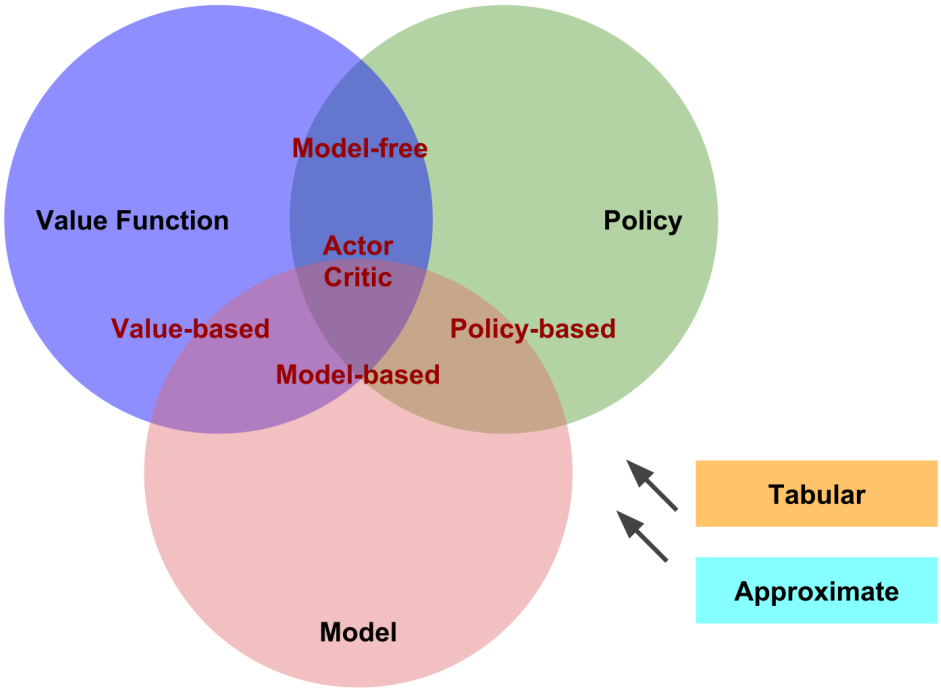
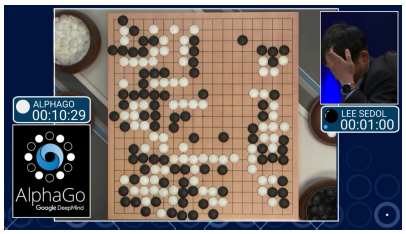


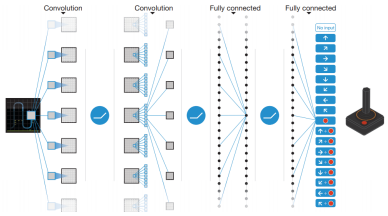
Figure 5: A rough categorization of RL algorithms

1.7 Notable Applications of Reinforcement Learning

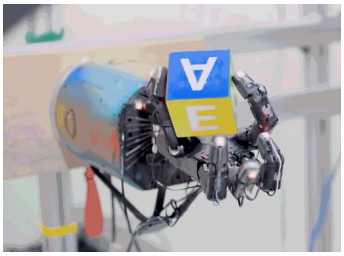
Some notable applications of RL are listed below (view the papers or check the URLs to know more):



(a) AlphaGo [8] [https://deepmind.com/research/alphago/]



(b) Deep Q Network [4] [https://youtu.be/W2CAghUiofY]



(c) Dexterous Object Manipulation [5] [https://youtu.be/jwSbzNHGflM?t=38s]

(d) Ball in a Cup [7] [https://youtu.be/Fhb26WdqVuE?t=47s]

(e) Autonomous flying [1] [https://www.youtu.be/VCdxqn0fcnE]

Figure 6: Notable Applications of Reinforcement Learning

2 Markov Decision Process

An agent being trained to perform a task by interacting with its environment must have the capabilities for (i) sensing the state of the environment (ii) taking actions for affecting the state (iii) realizing goals related to the state of the environment. A Markov Decision Process (MDP) is a formal mathematical framework that is used to define the interation between the learning agent and its environment in terms of states, actions and rewards.

2.1 Definition

For an agent interacting with its environment over a sequence of discrete timesteps, as shown in Fig. 7, an MPD is formally defined as shown below:

Markov Decision Process

A Markov Decision Process (MDP) is formally defined as the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, r, \gamma \rangle$, where:

- $\mathcal{S}$ is a finite set of states.
- $\mathcal{A}$ is a finite set of actions.
- $\mathcal{P}$ is a state transition probability function, which defines the probability of transitioning to the next state S_{t+1} from the current state S_t on taking the action A_t.
$$\mathcal{P}(s'|s,a) = \mathbb{P}[S_{t+1} = s'|S_t = s, A_t = a]$$
- r is a reward function, which defines the expected reward to be received on taking a particular action in a given state.
$$r(s,a,s') = \mathbb{E}[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s']$$
- γ is a discount factor for assigning more importance to immediate rewards.

Consider an agent interacting with its environment at each of a sequence of discrete timesteps $t = 0, 1, 2, 3, ...$ as shown in Fig. 7.

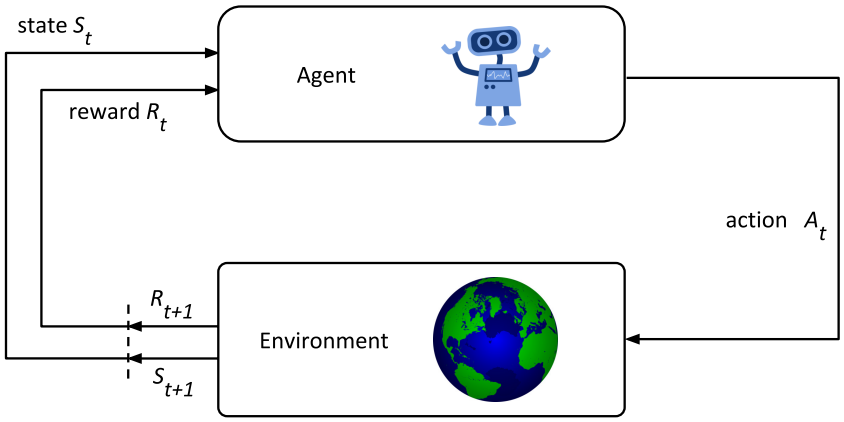


Figure 7: The reinforcement learning loop with states, rewards and action.

At each timestep t, the agent receives the current state of the environment $S_t \in \mathcal{S}$, and based on this state selects an action $A_t \in \mathcal{A}$. One timestep later, at $t+1$,

the agent receives the reward $R_{t+1} \in \mathbb{R}$ and the new environment state $S_{t+1} \in \mathcal{S}$. This gives rise to a *trajectory* (sequence of encountered states, actions and rewards) $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...$, as shown in Fig. 8. The question now is that if the agent at time t sees the state S_t and needs to decide on the action A_t, does it need to consider all the states, actions and rewards that happened before time t?

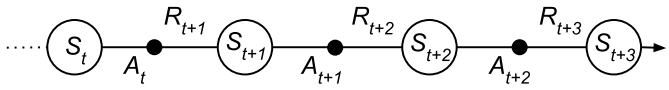


Figure 8: Sequence of states, actions and rewards.

One of the most important assumptions that RL algorithms rely on is the assumption of states having the *Markov Property*. This assumption is about the current state being enough for the agent to base its action on, and is the basis on which the theoretical foundation of many RL algorithms lies.

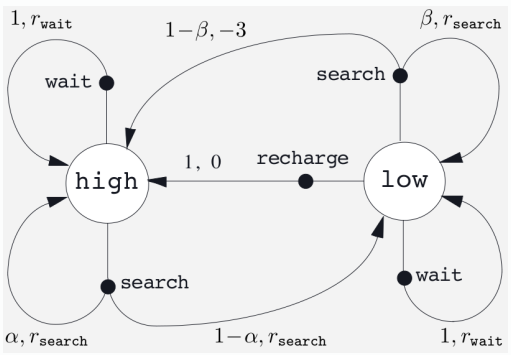
Markov Property

A state is said to possess the **Markov Property** when it includes information about all aspects of the past agent-environment interaction that make a difference for the future. In other words, if the present state is known, then we can make future decisions based on this state without needing to worry about the previous states, rewards and actions.

Example: Recycling Robot (example 3.3 in [10])

Consider a mobile robot that has the task of collecting empty cans in an office environment. It can perform 3 actions: (i) *search* for an empty can by moving about (ii) *wait* for someone to give it a can (iii) *recharge* its battery by moving to the charging station. The decision to take a particular action is taken based on the 2 possible states of its battery: (i) *low* (ii) *high*. Thus the set of states is $\mathcal{S} = \{low, high\}$ and the set of actions is $\mathcal{A} = \{search, wait, recharge\}$. Moreover, when the battery is *high*, the robot will not try to *recharge* its battery unnecessarily. Thus, actions in the state *high* are denoted as $\mathcal{A}(high) = \{search, wait\}$. Similarly, actions in the *low* state are $\mathcal{A}(low) = \{search, wait, recharge\}$. If we have the transition probabilities of switching between states and the corresponding rewards (as shown in the figure below), then we have the definition of the complete MDP for this scenario. The transition probabilities and rewards can also be summarized in the form of a table.

s	a	s'	$p(s'\|a,s)$	$r(s,a,s')$
high	*search*	*high*	α	r_{search}
high	*search*	*low*	$1-\alpha$	r_{search}
low	*search*	*high*	$1-\beta$	-3
low	*search*	*low*	β	r_{search}
high	*wait*	*high*	1	r_{wait}
high	*wait*	*low*	0	-
low	*wait*	*high*	0	-
low	*wait*	*low*	1	r_{wait}
low	*recharge*	*high*	1	0
low	*recharge*	*low*	0	-



Exercise: Markovian and Non-Markovian Env.

1. Devise an example task that fits into the MDP framework, identifying for each its states, actions, and rewards.
2. Can you think of an environment in which states do not have the Markov property?

2.2 Goals and Rewards

As stated previously, the **reward** $R_t \in \mathbb{R}$ is a scalar quantity that forms the basis of evaluating the action taken by an agent. The **goal** of an RL agent is to maximize the cumulative reward over the long run.

Any goal can be thought of as the maximization of the expected value of the cumulative reward. However, we should be careful about designing the reward so that it specifies what needs to be achieved, but not how to achieve it. For example, in chess, capturing opponent peices can be thought of as a sub-strategy. However, we should be careful about assigning rewards for capturing pieces, since the agent may start to prefer taking pieces than the final objective of winning (it may choose to take a piece even at the risk of sacrificing the king!).

The agent must be able to measure how well it is performing frequently over its lifespan. If rewards are **sparse** (for example, a robot travelling through a maze gets a reward of +1 only on finding a way out, and -1 otherwise, but nothing in between), the robot may have difficulty in evaluating its individual actions that led to success or failure.

Exercise: Maze Runner

What is a good choice of rewards for a maze solving agent?

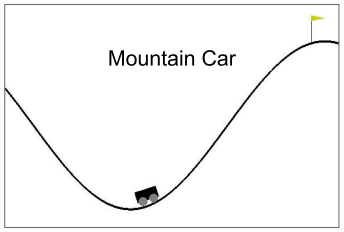
2.3 Episodes and Returns

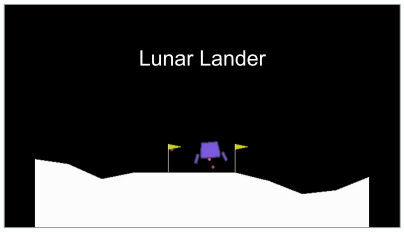
Reinforcement learning tasks can be of two types:

- **Episodic Tasks**: The agent-environment interaction breaks down naturally into subsequences known as episodes, which end with a terminal state S_T at timestep T. Once the terminal state has been reached, the agent's state is reset to its initial state S_0 and a new episode begins where the agent again starts interacting with the environment. In an MDP, all non-terminal states are denoted by the set $\mathcal{S}$ and the set containing all states (terminal and non-terminal) is denoted by $\mathcal{S}^+$.

Examples of Episodic Tasks

Shown below are two episodic tasks from the OpenAI Gym toolkit [4]





- **Continuing Tasks**: The agent-environment interaction does not break down in clearly identifiable sub-sequences, but carries on indefinitely ($T = \infty$).

Examples of Continuinig Tasks

Gas pipeline monitoring, heating system regulator for a large building.

For episodic tasks, if the agent expects to receive rewards $R_{t+1}, R_{t+2}, R_{t+3}, ..., R_T$ from time t onwards till time T, the **return** G_t is defined as:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T = \sum_{k=0}^{T} R_{t+k+1} \tag{1}$$

However, for continuing tasks, $T = \infty$ and so G_t may also evaluate to ∞ (assuming rewards are positive). In order to avoid this (and also for mathematical convenience), a discounting factor $\gamma \in [0, 1)$ is employed to limit the value of G_t to a finite quantity. By using γ, we assign less importance to rewards that can occur far into the future and more importance to rewards that are more immediately expected. Thus, by using the discounting factor γ, Eq. 1 is modified into

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2}$$

If we choose $\gamma = 0$, then only the next immediate reward is considered (myopic or short-sighted behavior). When $\gamma \approx 1$, all future rewards are given almost equal importance. The relationship between returns at successive steps can be easily derived:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + ...) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{3}$$

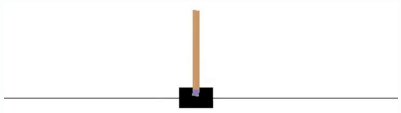
Exercise: Calculate the Return

Suppose γ= 0.5 and the following sequence of rewards is expected $R_1 = 1, R_2 = 2, R_3 = 6, R_4 = 3$, and $R_5 = 2$, with $T = 5$. What are $G_0, G_1, ..., G_5$? Hint: Work backwards.

[4] `https://gym.openai.com/envs`

Example: Pole Balancing

The task of pole balancing or cartpole [5] can be set up as an episodic task as well as a continuing task.



- **Episodic Task** (undiscounted): reward = +1 for each step before failure, return at each step = # steps to failure
- **Continuing Task**: reward = -1 on failure, 0 otherwise , return at each step is related to $-\gamma^k$, for k steps before failure

In both cases return is maximized by avoiding failure as long as possible.

2.4 Unified Notation for Episodic and Continuing Tasks

Episodic tasks can be viewed as a special case of continuing tasks where the terminal state state S_T acts as an absorbing state for which the reward is always 0 for any action taken in that state, as shown in Fig. 9.

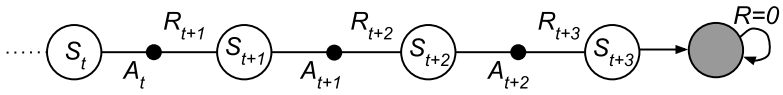


Figure 9: Unified notation for episodic and continuing tasks.

Thus, we can cover both continuing and episodic tasks by writing

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{4}$$

where γ can be 1 only if a zero-reward absorbing state is always reached.

2.5 Value Functions and Policies

As briefly mentioned in Section 1.6, value-based reinforcement learning algorithms estimate value functions (functions of state or state-action pairs). Value functions are defined in terms of the expected return ($\mathbb{E}[G_t]$) and denote how desirable it is to be in a given state, or how desirable it is to be in a given state and take a particular action.

Value functions are defined with respect to a policy (that is a particular way of acting or the behavior of the agent) (see Table 1).

Policy

A policy defines the behavior of an agent. It is a function which gives the mapping from states to the probabilities of selecting an action.

$$pi(a|s) = \mathbb{P}[A_t = a|S_t = s] \tag{5}$$

[5] `https://gym.openai.com/envs/CartPole-v1/`

If we have a policy π (a way of behaving), then it is possible to use value functions to figure out how good it is to be in a particular state if the same policy π is followed in the future. Accordingly, two different kinds of value functions can be defined.

State-value Function

The state-value function of a state s under a policy π is defined as the expected return when starting in s and following π thereafter.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s] \ \forall s \in \mathcal{S} \tag{6}$$

Action-value Function

The action-value function of a state s and action a under a policy π is defined as the expected return when starting in s, taking the action a (which may not necessarily be predicted by π) and following π thereafter.

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \ \forall s \in \mathcal{S}, a \in \mathcal{A} \tag{7}$$

Table 2 summarizes the descriptions and mathematical expressions for the important terminology discussed so far.

Table 2: Summary of Terms

Term	**Description**	**Expression**
MDP	Framework defining agent-environment interaction	$\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, r, \gamma \rangle$ where $\mathcal{S}$: finite set of discrete states $\mathcal{A}$: finite set of discrete actions $\mathcal{P}$: state transition probability function r: reward function γ: discount factor.
Markov Property	Current state includes all information about the past	-
Reward	Scalar quantity for evaluating the agent's action.	$R_t \in \mathbb{R}$
Return	Discounted sum of future rewards.	$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ $= R_{t+1} + \gamma G_{t+1}$
Goal	Maximize expected Return	$maximize(\mathbb{E}[G_t])$ at each t

Table 2: Summary of Terms

Term	Description	Expression
Policy	Mapping from states to probabilities of actions.	$\pi(a\|s) = \mathbb{P}[A_t = a\|S_t = s]$
State-value Function	Expected **Return** when starting in s and following π thereafter.	$v_\pi(s) = \mathbb{E}_\pi[G_t\|S_t = s]$ $= \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\|S_t = s]\ \forall s \in \mathcal{S}$
Action-value Function	Expected **Return** when starting in s, taking action a and following π thereafter.	$q_\pi(s,a) = \mathbb{E}_\pi[G_t\|S_t = s, A_t = a]$ $= \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\|S_t = s, A_t = a]$ $\forall s \in \mathcal{S}, a \in \mathcal{A}$

2.6 Bellman Expectation Equations

A fundamental property of value functions is that they satisfy recursive relationships between values of current and next states (similar to the relationship for the expected return in Eq. 3).

The **Bellman Expectation equation** for v_π expresses a relation between the value of a state and the values of its successor states. This is a system of linear equations (one for each state in the case of v_π, or one for each state-action pair in the case of q_π), which has a unique solution.

The state-value function of a state s is defined as the expectation over the returns G_t starting from state $S_t = s$:

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi\Big[G_t \;\Big|\; S_t = s\Big] \text{ (by definition)} \\
&= \mathbb{E}_\pi\Big[R_{t+1} + \gamma G_{t+1} \;\Big|\; S_t = s\Big] \text{ (using Eq. 3)} \\
&= \mathbb{E}_\pi\Big[R_{t+1} + \gamma \mathbb{E}_\pi\big[G_{t+1} \;\big|\; S_{t+1} = s'\big] \;\Big|\; S_t = s\Big] \text{ (using the law of total expectation)}^6 \\
&= \mathbb{E}_\pi\Big[R_{t+1} + \gamma v_\pi(s') \;\Big|\; S_t = s\Big] \text{ (inner } \mathbb{E} \text{ gives value of the successor state } s') \qquad (8) \\
&= \sum_a \pi\big(a \;\big|\; s\big) \sum_{s',r} p\big(s', r \;\big|\; s, a\big)\big[r + \gamma v_\pi(s')\big] \quad \forall s, s' \in \mathcal{S}\ a \in \mathcal{A} \qquad (9) \\
&\qquad \text{(replacing the expectation with summations over probabilities)}
\end{aligned}
$$

The Bellman equation (Eq. 9) using the summation over probabilities can be convieniently represented by using a **backup diagram**, as shown in Fig. 10. A backup diagram shows all states, actions and rewards that participate in a backup (update of a value based on downstream values). In the diagram, time flows from top to bottom and states are not necessarily distinct (a backup diagram is not a transition graph).

[6] `https://en.wikipedia.org/wiki/Law_of_total_expectation`

Fig. 10 (left) shows the backup diagram for the state-value function. The white circle at the top represents the current state s and the white circles at the bottom represent the successor states s' that can be reached by taking an action a according to the policy π. Actions are represented by the black dots and the arrows represent the transitions (only a few possible actions/states are shown). Once we have the values of the lower states, then they can be combined using Eq. 9 to obtain the value of the upper state.

Note that for a state, the choice of an action rests with the agent (controlled by policy π shown in red), but when that action is taken, the resulting state that is produced depends completely on the environment (based on the dynamics of the environment - probability p shown in blue).

The inner summation in Eq. 9 sums up the discounted values of each successor state s' plus the reward multiplied by the probability of reaching s'. These values are then multiplied by the probability of taking the action a that may lead to s' and summed up using the outer summation in Eq. 9.

Bellman Expectation Equation for state-value functions

The Bellman Expectation equation for the state-value function expresses a relation between the value of a state s and the values of its successor states s'. This is a system of linear equations (one for each state), which has a unique solution.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi\Big[R_{t+1} + \gamma v_\pi(s') \;\Big|\; S_t = s\Big] \\ &= \sum_a \pi\big(a \mid s\big) \sum_{s',r} p\big(s', r \mid s, a\big)\big[r + \gamma v_\pi(s')\big] \;\; \forall s, s' \in \mathcal{S} \;\; a \in \mathcal{A} \end{aligned}$$

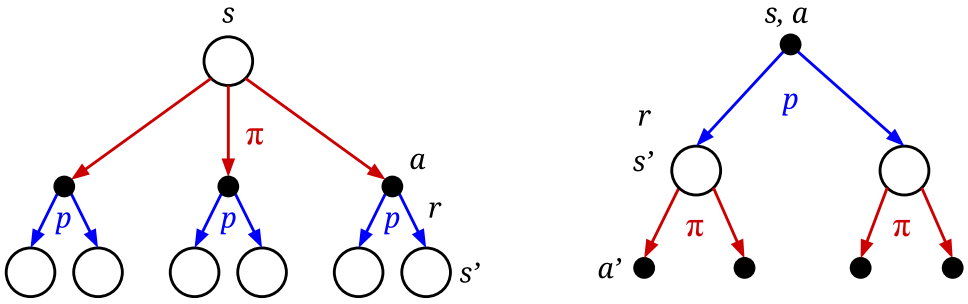


Figure 10: Backup diagrams for the Bellman Expectation equations for the (left) state-value function (right) action-value function.

Similar to the Bellman Expectation Equation for the state-value function, there is also a similar Bellman equation for the action-value function, which represents the recursive relationship between action-values of state-action pairs and their successors. The corresponding backup diagram is shown in Fig. 10 (right).

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi\Big[G_t \;\Big|\; S_t = s, A_t = a\Big] \\ &= \mathbb{E}_\pi\Big[R_{t+1} + \gamma G_{t+1} \;\Big|\; S_t = s, A_t = a\Big] \\ &= \mathbb{E}_\pi\Big[R_{t+1} + \gamma \mathbb{E}_\pi\big[G_{t+1} \;\big|\; S_{t+1} = s', A_{t+1} = a'\big] \;\Big|\; S_t = s, A_t = a\Big] \\ &= \mathbb{E}_\pi\Big[R_{t+1} + \gamma q_\pi(s', a') \;\Big|\; S_t = s, A_t = a\Big] \qquad (10) \\ &= \sum_{s'} p\big(s' \mid s, a\big)\Big[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')\Big] \;\; \forall s, s' \in \mathcal{S} \; a, a' \in \mathcal{A} \qquad (11) \end{aligned}$$

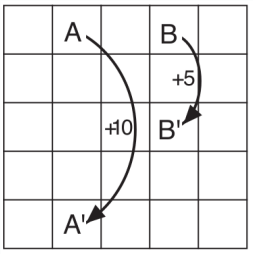
Bellman Expectation Equation for action-value functions

The Bellman Expectation equation for the action-value function expresses a relation between the action-value of a state s and action a and the values of its successor states s' and actions a'. This is a system of linear equations (one for each state-action pair), which has a unique solution.

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi\Big[R_{t+1} + \gamma q_\pi(s',a') \;\Big|\; S_t = s\Big] \\ &= \sum_{s'} p\big(s' \mid s,a\big)\Big[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s',a')\Big] \quad \forall s,s' \in \mathcal{S}\ a,a' \in \mathcal{A} \end{aligned}$$

Example: Gridworld (example 3.5 in [10])

The following figure shows an example where the Bellman expectation equation is used recursively to compute the value function of a given policy in a gridworld.



3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

- **Rewards**: -1 for going off the edge; 0 otherwise except for the special cases shown
- **Discount Factor**: $\gamma = 0.9$
- **Environment**: Deterministic as shown
- **Policy**: Uniform random
- Negative values near edge, A's return less than immediate reward and B's is greater than immediate reward

Exercise: 2-State Random Walk (solved)

Calculate the state-value function of the states A and B (assuming policy π chooses actions randomly) using the Bellman expectation equation for the folowing scenario:

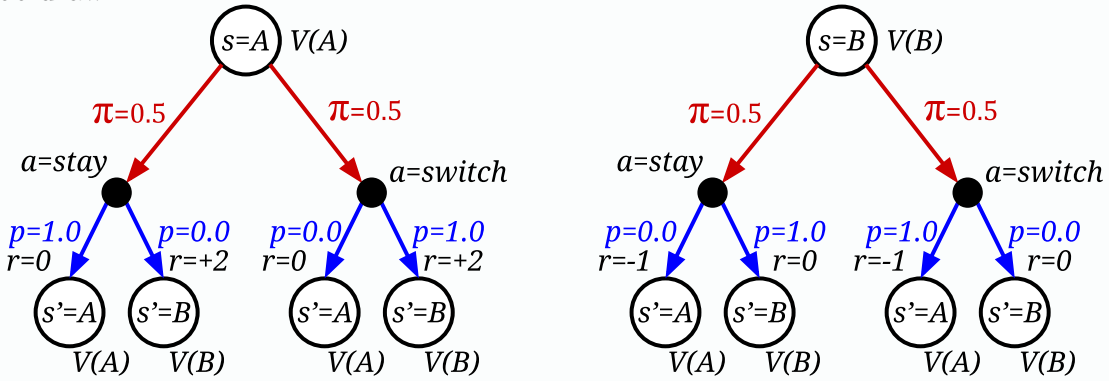
- **States**: A and B
- **Actions**: stay or switch (equal probabilities, deterministic effect)
- **Rewards**: +2 for A→B; -1 for B→A; 0 otherwise
- **Discount Factor**: $\gamma = 0.9$

(solution on next page)

Exercise: 2-State Random Walk (solved)

Solution:

An easy way to work out problems such as this is to use the backup diagrams. For every distinct state, a backup diagram is drawn with that state as the root. For example, for the states A and B respectively as root, the following diagrams can be drawn:



Each of these diagrams will give us a linear equation where the unknowns are V(A) and V(B). Therefore we will end up with 2 equations in 2 unknowns.

Writing down the Bellman expectation equation again:

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_\pi(s')\right] \; \forall s, s' \in \mathcal{S} \; a \in \mathcal{A}$$

For the diagram on the left ($s = A$), filling in the values of p, r and γ, we see that the inner summation over probabilities for $a = stay$ can be written as $\{1.0(0 + 0.9V(A)) + 0.0(2 + 0.9V(B))\}$ and the inner summation for $a = switch$ can be written as $\{0.0(0 + 0.9V(A)) + 1.0(2 + 0.9V(B))\}$. Since the probability for either action is 0.5, the outer summation over the 2 actions can be written out to get $V(A)$. Thus:

$$\begin{aligned} V(A) &= 0.5\{(1.0(0 + 0.9V(A)) + 0.0(2 + 0.9V(B))\} \\ &\quad + 0.5\{(0.0(0 + 0.9V(A)) + 1.0(2 + 0.9V(B))\} = 0.45V(A) + 0.45V(B) + 1.0 \\ &\Rightarrow 0.55V(A) - 0.45V(B) = 1 \quad \text{(eq. I)} \end{aligned}$$

Performing the same steps for the diagram on the right ($s = B$), we get:

$$\begin{aligned} V(B) &= 0.5\{(0.0(-1 + 0.9V(A)) + 0.1(0 + 0.9V(B))\} \\ &\quad + 0.5\{(1.0(-1 + 0.9V(A)) + 0.0(0 + 0.9V(B))\} = 0.45V(B) + 0.45V(A) - 0.5 \\ &\Rightarrow 0.45V(A) - 0.55V(B) = 0.5 \quad \text{(eq. II)} \end{aligned}$$

$$\text{((eq. II)} \div 0.45) - \text{((eq. II)} \div 0.55) \Rightarrow \frac{11}{9}V(A) - \frac{9}{11}V(A) = \frac{20}{9} - \frac{10}{11} \Rightarrow V(A) = 3\frac{1}{4}$$

Solving for $V(B)$, we get $V(B) = 1\frac{3}{4}$

2.7 Optimal Policies and Value Functions

The task of RL is to find a policy that achieves the maximum possible reward in the long run. By using the Bellman Expectation equations (Eq. 8 - 11), it is possible to evaluate the value of a given policy. This makes it possible to define a **partial ordering over policies** (comparing policies according to their value functions). A policy π is considered to be better than a policy π':

$$\pi > \pi' \text{ if and only if } v_\pi(s) \geq v'_\pi(s) \ \forall s \in \mathcal{S} \tag{12}$$

The optimal policy π_* (best possible policy) is better than or equal to all other policies. Consequently, the optimal state-value (or action-value) function v_* is the maximum state-value (action-value) function among all policies.

$$v_*(s) = \max_\pi v_\pi(s) \ \forall s \in \mathcal{S} \tag{13}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a) \ \forall s \in \mathcal{S} \ a \in \mathcal{A} \tag{14}$$

2.8 Bellman Optimality Equations

The Bellman Optimality Equation for the optimal value-function v_* expresses the fact that the value of a state under an optimal policy must be equal to the expected return for the best action from that state.

$$\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\
&= \max_a \ \mathbb{E}_{\pi_*}\big[G_t \ \big| \ S_t = s, A_t = a\big] \\
&= \max_a \ \mathbb{E}_{\pi_*}\big[R_{t+1} + \gamma G_{t+1} \ \big| \ S_t = s, A_t = a\big] \text{ (using Eq. 3)} \\
&= \max_a \ \mathbb{E}_{\pi_*}\big[R_{t+1} + \gamma v_*(S_{t+1}) \ \big| \ S_t = s, A_t = a\big] \qquad (15) \\
&= \max_a \sum_{s',r} p\big(s', r \ \big| \ s, a\big)\big[r + \gamma v_*(s')\big] \qquad (16)
\end{aligned}$$

(removing the expectation by summing over states and rewards)

In the backup diagram shown in Fig. 11 (left), the max operator operates over all the possible actions that can be taken from state s, and the summation is performed for all possible states and rewards that can follow the taking of action a (compare with Eq. 16).

Bellman Optimality Equation for v_*

$$\begin{aligned}
v_*(s) &= \max_a \ \mathbb{E}_{\pi_*}\big[R_{t+1} + \gamma v_*(S_{t+1}) \ \big| \ S_t = s, A_t = a\big] \\
&= \max_a \sum_{s',r} p\big(s', r \ \big| \ s, a\big)\big[r + \gamma v_*(s')\big]
\end{aligned}$$

Similarly, there is a Bellman Optimality Equation for the optimal action-value function q_* also (Eq. 17 and 18). The corresponding backup diagram (Fig. 11 (right)) shows

that the summation in Eq. 18 is performed over all possible state transitions on taking action a and the max is performed over the possible successor actions a'.

Bellman Optimality Equation for q_*

$$q_*(s,a) = \mathbb{E}\big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\big|\; S_t = s, A_t = a\big] \tag{17}$$

$$= \sum_{s',r} p\big(s', r \;\big|\; s, a\big)\big[r + \gamma \max_{a'} q_*(s', a')\big] \tag{18}$$

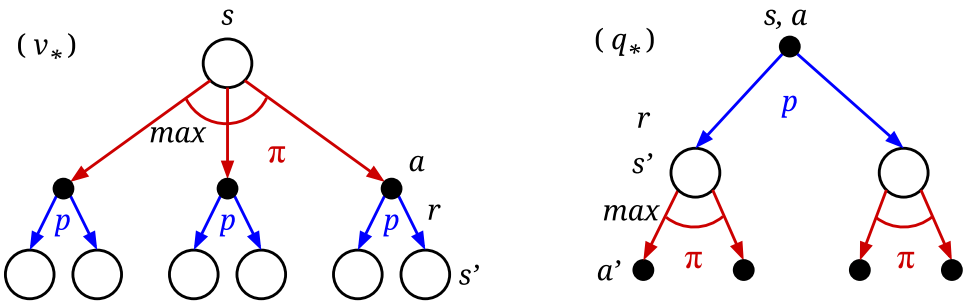


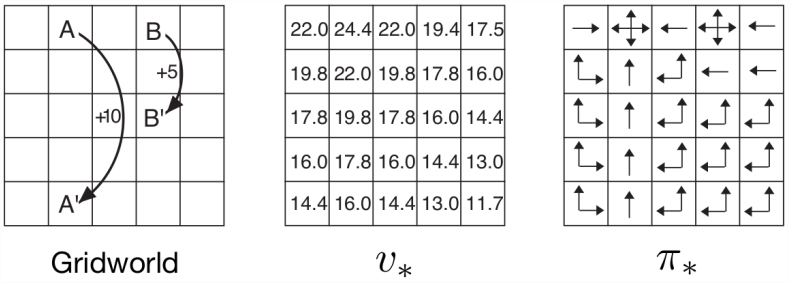
Figure 11: Backup diagrams for the Bellman Optimality equations for (left) v_* (right) q_*.

The Bellman Optimality equations are a system of non-linear equations (due to the use of the max operator). If the environment dynamics p is known, then an exact solution can be obtained. Once we have the optimal value function v_*, then the actions which appear best after a one-step search (looking at the values of the successor states and choosing the action that leads to the successor with the highest value) will lead to optimal actions (although this looks like a greedy strategy, v_* already contains information about the future). If we have the optimal action-value function q_*, then even a one-step search is not necessary. For any state s, the agent can simply find the action a that results in the maximum value (maximizes $q_*(s,a)$).

Determining π_* from v_* or q_*

- v_* is known: Take an action that is greedy w.r.t. v_* (search through actions)
- q_* is known: In state s take the action a that has the maximum value of $q_*(s,a)$ (no search is necessary)

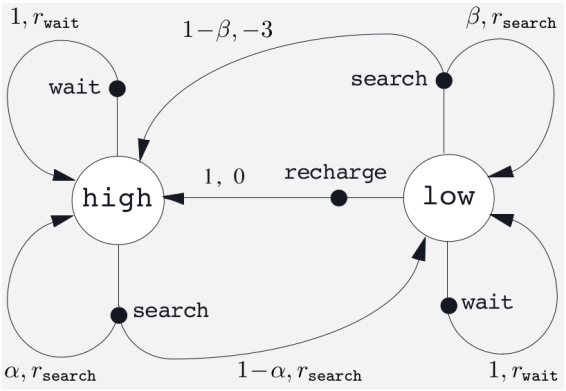
For example, if v_* for the gridworld is known, π_* can be easily calculated by choosing the action that leads to the state with the highest value according to v_*.



Exercise: v_* for Recycling Robot

Write down the Bellman optimality equation(s) for v_* for the recycling robot.

$$v_*(s) = \max_a \sum_{s',r} p\big(s', r \mid s, a\big)\big[r + \gamma v_*(s')\big]$$



3 Dynamic Programming

Dynamic Programming (DP) is a technique for solving complex problems by (i) breaking the problem down to smaller sub-problems, and (ii) solving the sub-problems and combining the results.

DP assumes full knowledge of the MDP (environment dynamics), and hence is of limited utility in practice. However, it is still theoretically very important because it forms the basis for other techniques which try to approximately achieve the effects of DP with less computation and without assuming perfect knowledge of the environment.

In DP, we assume a finite MDP (Markov Decision Process) in which the sets of states and actions are finite. For continuous state and action spaces, these spaces can be quantized by using techniques such as **tile coding** and then the finite-state DP methods can be applied.

Prediction and Control

A key idea of Dynamic Programming is to use value functions to organize and structure the search for good policies. Two types of problems are handled: predicting the value function for an existing policy and determining the optimum policy.

Prediction
Problem: Calculate the value function for a policy
Input: MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, r, \gamma \rangle$ and policy π
Output: Value function v_π

Control
Problem Calculate the optimal policy
Input: MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, r, \gamma \rangle$
Output: Optimal value function v_* and optimal policy π_*

3.1 Policy Evaluation

Policy evaluation refers to the prediction problem stated above.

Problem Compute the state-value function v_π for an arbitrary policy π.

Solution Iterative application of the Bellman Expectation backup using synchronous backups (all states updated every time).

For all states $s \in \mathcal{S}$, at each iteration $k+1$, update $v_{k+1}(s)$ from $v_k(s)$ using

$$v_{k+1}(s) = \mathbb{E}_\pi \big[R_{t+1} + \gamma v_k(S_{t+1}) \;\big|\; S_t = s \big] \tag{19}$$

$$= \sum_a \pi\big(a \;\big|\; s\big) \sum_{s',r} p\big(s', r \;\big|\; s, a\big) \big[r + \gamma v_k(s') \big] \;\; \forall s \in \mathcal{S} \tag{20}$$

Starting from an initial value v_0, repeated updates as shown above results in a sequence of value functions which converge to v_π as k approaches ∞ (in practice sooner than that):

$$v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_\pi$$

When the change in the updated value is negligible, the value evaluation process is said to have converged.

Algorithm 1 Iterative Policy Evaluation for estimating $V \approx v_\pi$

Input: π, the policy to be evaluated
Parameter: $\theta > 0$, a small threshold for determining the accuracy of estimation
Initialize: $V(s)\ \forall s \in \mathcal{S}^+$, a table used for approximating v_π. Initialization can be arbitrary (or set everything to 0), except for $V(terminal) = 0$
repeat
  $\Delta \leftarrow 0$
  for each $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi\big(a \mid s\big) \sum_{s',r} p\big(s', r \mid s, a\big)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, \mid v - V(s) \mid)$
  end for
until $\Delta < \theta$

Exercise: Policy Evaluation Algorithm

For the gridworld example discussed on pg. 17 of this document (example 3.5 in [10]), the final value-function for the uniform random policy is shown below. Verify that the value of the state V(row:2,col:2)=0.71 does not change appreciably using the update step in the Policy Evaluation algorithm:

$$V(s) \leftarrow \sum_a \pi\big(a \mid s\big) \sum_{s',r} p\big(s', r \mid s, a\big)\big[r + \gamma V(s')\big]$$

3.36	8.83	4.47	5.36	1.53
1.57	3.04	2.29	1.95	0.59
0.10	0.78	0.71	0.40	-0.37
-0.93	-0.39	-0.32	-0.55	-1.15
-1.81	-1.30	-1.19	-1.39	-1.94

3.2 Policy Improvement

The reason why we want to evaluate policies is for comparing policies and eventually finding better policies. For a given state s, we would like to find out whether or not we should change the policy π by deterministically choosing an action $a \neq \pi(s)$ and then to follow π thereafter.

The value of the policy π is v_π. If we take the action $a \neq \pi(s)$ and then follow π, then the action-value of π is:

$$q_\pi(s,a) = \mathbb{E}\big[R_{t+1} + \gamma v_\pi\big(S_{t+1}\big) \;\big|\; S_t = s, A_t = a\big] \tag{21}$$

Now if $v_\pi(s) \leq q_\pi(s,a)$, then it is better to follow the modified policy π' (take action a whenever state s is encountered). It can be shown that if $v_\pi(s) \leq q_\pi(s,a)$, then $v_\pi(s) \leq v'_\pi(s)$ (value of the modified policy is more)(proof on pg. 78 of [10]).

Thus, in order to improve the policy π, it is natural to act greedily at each step and choose the policy that gives the best value. In other words, a better, modified policy π' can be obtained from π by:

$$\begin{aligned}\pi'(s) &= \arg\max_a q_\pi\big(s,a\big) \\ &= \arg\max_a \mathbb{E}\big[R_{t+1} + \gamma v_\pi\big(S_{t+1}\big) \;\big|\; S_t = s, A_t = a\big]\end{aligned} \tag{22}$$

Suppose the new policy π' is as good as π but not better than it. Then $v'_\pi = v_\pi$. So, it follows that

$$v'_\pi(s) = \max_a \mathbb{E}\big[R_{t+1} + \gamma v_\pi\big(S_{t+1}\big) \;\big|\; S_t = s, A_t = a\big]$$

which is the same as the Bellman Optimality Equation (Eq. 15). Thus, when $v'_\pi = v_\pi$, both v'_π and v_π must be the optimal value function v_*, and π' and π must be the optimal policy π_*.

3.3 Policy Iteration

Policy iteration is the process of generating monotonically improving policies and value functions by alternative use of policy evaluation (Sec. 3.1) and policy improvement (Sec. 3.2). This process ultimately converges to the optimal policy and value function.

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \ldots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$

where E refers to policy evaluation and I refers to policy improvement. This process is depicted in Fig. 12.

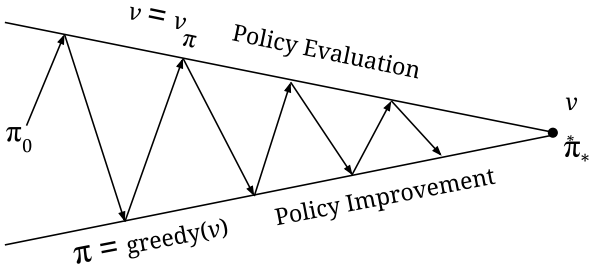


Figure 12: Policy Iteration.

Algorithm 2 Policy Iteration for estimating $V \approx v_*$ and $\pi \approx \pi_*$

1. **Initialization**
   $V(s) \in \mathbb{R}$ and $\pi(s) = \mathcal{A}\ \forall s \in \mathcal{S}$
2. **Policy Evaluation**
   Input: π, the policy to be evaluated
   Parameter: $\theta > 0$, a small threshold for determining the accuracy of estimation
   Initialize: $V(s)\ \forall s \in \mathcal{S}^+$
   repeat
   $\Delta \leftarrow 0$
   for each $s \in \mathcal{S}$ **do**
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_a \pi\big(a \mid s\big) \sum_{s',r} p\big(s', r \mid s, a\big)\big[r + \gamma V(s')\big]$
   $\Delta \leftarrow \max(\Delta, \mid v - V(s) \mid)$
   end for
   until $\Delta < \theta$
3. **Policy Improvement**
   policy-stable $\leftarrow$ *true*
   for each $s \in \mathcal{S}$ **do**
   old-action $\leftarrow \pi(s)$
   $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   end for
   If *policy-stable*, then stop and return $V \approx v*$ and $\pi \approx \pi_*$; else go to 2.

Exercise: Policy Iteration Algorithm

The policy iteration algorithm has a subtle bug in that it may never terminate if the policy continually switches between two or more policies that are equally good.

Can you modify the pseudocode so that convergence is guaranteed?

3.4 Value Iteration

Although the policy iteration algorithm (Algo. 2) can compute the optimal policy, the disadvantage is that it involves the process of policy evaluation (Algo. 1), which itself is an iterative process.

Value Iteration turns the Bellman optimality equation (Eq. 15 and 16) into an update rule, without needing to evaluate intermediate policies iteratively.

$$v_{k+1}(s) = \max_a \mathbb{E}\big[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a\big] \tag{23}$$

$$= \max_a \sum_{s',r} p\big(s', r \mid s, a\big)\big[r + \gamma v_k(s')\big]\ \forall s \in \mathcal{S} \tag{24}$$

For any arbitrary initialization v_0, this converges to v_*. The update in Eq. 23 is same as Policy Evaluation (Eq. 19) except that max is taken over all actions. Finally, the optimal policy π_* is obtained by acting greedily with respect to v_*.

Algorithm 3 Value Iteration for estimating $\pi \approx \pi_*$

Algorithm parameter $\theta > 0$
Initialize $V(s)$, $\forall s \in \mathcal{S}^+$ arbitrarily, except $V(terminal) = 0$
repeat
  $\Delta \leftarrow 0$
  for each $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p\big(s', r \bigm| s, a\big)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max\big(\Delta, |v - V(s)|\big)$
  end for
until $\Delta < \theta$
Output a deterministic policy $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p\big(s', r \bigm| s, a\big)\big[r + \gamma V(s')\big]$

Table 3 summarizes the 3 algorithms discussed in this section.

Table 3: Summary of Dynamic Programming algorithms

Problem	**Bellman Equation**	**Algorithm**
Prediction	Bellman Expectation Equation	Iterative Policy Evalutaion
Control	Bellman Expectation Equation + greedy policy improvement	Policy Iteration
Control	Bellman Optimality Equation	Value Iteration

4 Model-free Prediction and Control

Till now, for the task of finding the optimal way for the agent to behave, we have assumed that we know the MDP (that is we know what the state transition probability function $\mathcal{P}$ and the reward function r are). For example, in Sec. 2 and Sec. 3, the gridworld environments were completely deterministic and the rewards for transitioning between specific states was pre-defined. In the real world, however, we do not have the details of the MDP and so we must infer the environmental dynamics through repeated interactions.

Model-free Prediction and Control

Model-free prediction and control deals with the task of determining the value of a policy or of determining the optimal value function and the optimal policy when the MDP is not known by learning from experience.

4.1 Model-free Prediction

In dynamic programming, the value of a policy was evaluated using the iterative policy evaluation algorithm (Algo. 1) by utilizing the knowledge about the MDP. In model-free prediction, the same task is accomplished withouth the MDP, by learning from the agent's experience. This can be achieved by using 2 groups of methods: (i) Monte Carlo learning (MC-learning) and (ii) Temporal Difference learning (TD-learning).

4.1.1 Monte Carlo Learning

Monte Carlo learning involves learning the value function directly from complete episodes of experience (no bootstrapping) by using the average of the sampled returns as an estimate of the value of a state. This only applies to episodic tasks which have a terminal state S_T and is the simplest possible technique for performing model-free prediction.

Monte Carlo Policy Evaluation

- Learn the value function directly from episodes of experience.
- Learns from complete episodes (no bootstrapping) and is only applicable for episodic tasks.
- Uses the empirical mean return after an episode (instead of the expected return) as the value.

The goal of MC-learning for policy evaluation is to learn v_π from episodes of experience such as $S_0, A_0, R_1, S_1, A_1, R_2, ..., S_T \sim \pi$. The return at time t is defined as the discounted sum of rewards from t onwards: $G_t = \sum_{k=0}^{T} \gamma^k R_{t+k+1}$ (modified Eq. 4 by using T instead of ∞). Recall also, that the value function is defined as the expectation over the return $v_\pi = \mathbb{E}_\pi[G_t|S_t = s]$ (Eq. 6). MC-learning uses the empirical mean return (average of returns observed from roll-outs[7]) instead of this expected return. As more and more returns are observed, the average should converge to the expected value.

[7] A full, terminal episode is called a roll-out.

An algorithm for MC-learning is listed in Algo. 4. Here episodes are generated continuously. In each episode, the value of a state is updated with the average return whenever that state is encountered for the first time in that episode, leading to the name *First-visit* MC Prediction.

Algorithm 4 First-visit MC Prediction, for estimating $V \approx v_\pi$

Input:
  Policy π to be evaluated
Initialize
  $V(s) \in \mathbb{R},\ \forall s \in \mathcal{S}$ arbitrarily
  $Returns(s) \leftarrow$ an empty list $\forall s \in \mathcal{S}$
repeat
  Generate an episode using π: $S_0, A_0, R_1, S_1, A_1, R_2, ..., S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  for each step of episode, $t = T-1, T-2, ..., 0$ **do**
    $G \leftarrow R_{t+1} + \gamma G$
    if S_t does not appear in $S_0, S_1, .., S_{t-1}$ **then**
      Append G to $Returns(s)$
      $V(S_t) \leftarrow average(Returns(S_t))$
    end if
  end for
until forever

Instead of updating the value of the state only on its first occurance in an episode, we can also update the value whenever the state is encountered in the episode (by removing the **if** condition in Algo. 4). This would result in a modified algorithm known as *Every-visit* MC Prediction.

Another way of achieving the same effect is to update the value of $V(S_t)$ incrementally, instead of calculating the entire average over $Returns(S_t)$ every time.

Incremental Mean

The means $\mu_1, \mu_2, ...$ of a sequence $x_1, x_2, ...$ can be computed incrementally as:

$$\mu_k = \frac{1}{k}\sum_{j=1}^{k} x_j = \frac{1}{k}\Big(x_k + \sum_{j=1}^{k-1} x_j\Big) = \frac{1}{k}\Big(x_k + (k-1)\mu_{k-1}\Big) = \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1}) \quad (25)$$

We can maintain a counter $N(S_t)$ for each state which records the number of times the states have been visited. These counters are initialized to 0 and persist across episodes, and are incremented whenever (or the first time) the respective state is encountered during an episode. Using Eq. 25 and noting that $V(S_t)$ stores the average return, the step for updating $V(S_t)$, we can modify Algo. 4 slightly to the following, simpler algorithm.

Algorithm 5 Incremental First-visit MC Prediction, for estimating $V \approx v_\pi$

Input:
  Policy π to be evaluated
Initialize
  $V(s) \in \mathbb{R},\ \forall s \in \mathcal{S}$ arbitrarily
  $N(s) \in \mathbb{Z}$ an integer counter $\forall s \in \mathcal{S}$
repeat
  Generate an episode using π: $S_0, A_0, R_1, S_1, A_1, R_2, ..., S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  for each step of episode, $t = T-1, T-2, ..., 0$ **do**
    $G \leftarrow R_{t+1} + \gamma G$
    if S_t does not appear in $S_0, S_1, .., S_{t-1}$ **then**
      $N(S_t) \leftarrow N(S_t) + 1$
      $V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}\Big(G - V(S_t)\Big)$
    end if
  end for
until forever

For non-stationary problems, it is useful to track a running mean, by removing the counter $N(S_t)$ and instead using a constant α. Thus, the update step is written as follows.

Monte Carlo Policy Evaluation Update for learning $V \approx v_\pi$

$$V(S_t) \leftarrow V(S_t) + \alpha\big(G_t - V(S_t)\big) \tag{26}$$

Similary for learning the action-value function q_π, the MC update rule can be written as:

Monte Carlo Policy Evaluation Update for learning $Q \approx q_\pi$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big(G_t - Q(S_t, A_t)\big) \tag{27}$$

The approximate action-value function Q also converges to the true action-value function q_π if all state-action pairs are visited sufficiently often and a large number of episodes are generated. This can be helped by ensuring that all state-action pairs have a non-zero probability of being the starting pair (**exploring starts**).

4.1.2 Temporal Difference Learning

With MC-learning, we saw that it is possible to learn the value function without using an MDP. The downside is that we still need to wait for episodes to terminate and can only use MC-learning in episodic tasks.

In the previous section (Sec. 3), we saw that DP uses estimates of the value of successor states for updating the value function (bootstrapping $\Rightarrow$ make an estimate based on another estimate), but for this we need an MDP.

Temporal Difference learning (TD-learning) combines the best of both the above methods by allowing bootstrapping without needing a model. This allows us to update the

value function based on the estimated values of successor states at every step, without having to wait for an episode to terminate (which means that TD can also be used for continuing tasks). TD-learning updates the value of a state S based on the immediate next reward R and the estimated value of the successor state S' as shown below.

Temporal Difference Policy Evaluation Update for learning $V \approx V_\pi$

$$V(S) \leftarrow V(S) + \alpha\big(\underline{R + \gamma V(S')} - V(S)\big) \tag{28}$$

Note: The underlined part of the equation shows the **target**, an estimate of the expected return, and we update the value towards this target.

Using the above update step, we get the tabular algorithm for TD(0) or one-step TD (since an update is made after taking every action):

Algorithm 6 Tabular TD(0), for estimating $V \approx v_\pi$

Input: Policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize: $V(s) \in \mathbb{R}$, $\forall s \in \mathcal{S}^+$ arbitrarily, except $V(terminal) = 0$
for each episode **do**
  Initialize S
  for each step of episode until S is terminal **do**
    $A \leftarrow$ action taken by π for S
    Take action A, observe reward R and next state S'
    $V(S) \leftarrow V(S) + \alpha\big(R + \gamma V(S') - V(S)\big)$
    $S \leftarrow S'$
  end for
end for

For the action-value function too, there is a similar TD update.

Temporal Difference Policy Evaluation Update for learning $Q \approx q_\pi$

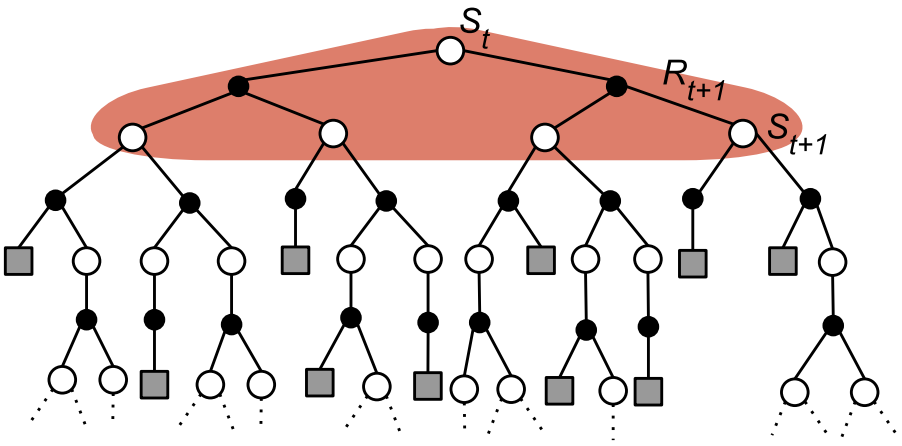
$$Q(S, A) \leftarrow Q(S, A) + \alpha\big(\underline{R + \gamma Q(S', A')} - Q(S, A)\big) \tag{29}$$

Note: The underlined part of the equation shows the **target**, an estimate of the expected return, and we update the value towards this target.

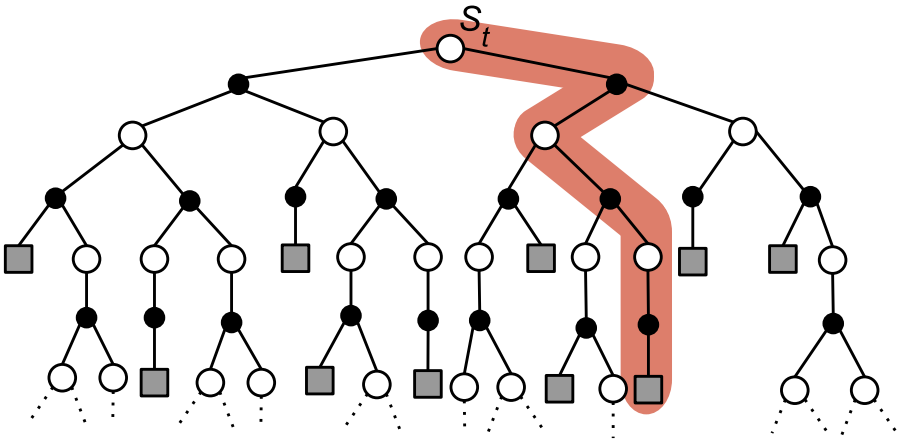
Now that we have looked at DP, MC and TD-learning, we can identify the major differences between these methods by using the backup diagrams shown in Fig. 13.

4.1.3 Advantages of TD Learning

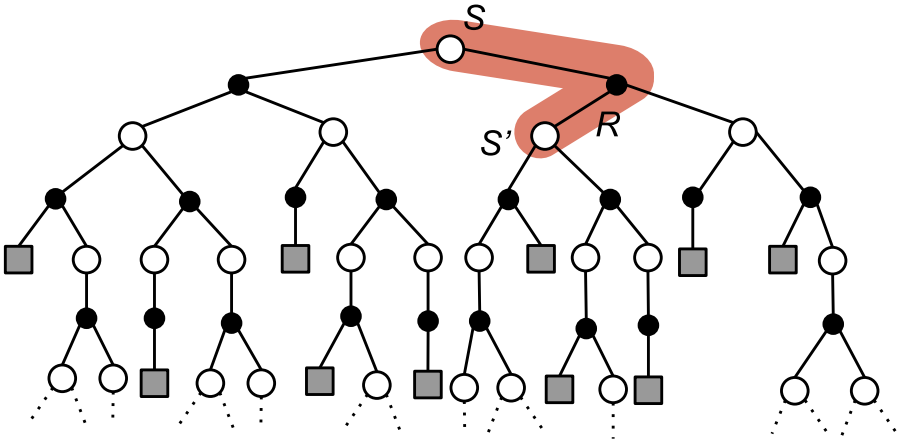
Similar to Fig. 13, Table 4 also shows the differences between DP, MC and TD. It can be seen that TD can learn without a model of the environment and also without requiring to play out an entire episode, and is therefore fully incremental:



(a) **DP**: Updates use $\mathbb{E}$ over all states/actions.
$V(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$



(b) **MC**: Updates use true return at the end of the episode.
$V(S_t) \leftarrow V(S_t) + \alpha\big(G_t - V(S_t)\big)$



(c) **TD(0)**: Updates are done after a single step.
$V(S) \leftarrow V(S) + \alpha\big(R + \gamma V(S') - V(S)\big)$

Figure 13: Backup diagrams to illustrate the differences between Dynamic Programming (DP), Monte Carlo learning (MC) and one-step Temporal Difference Learning (TD(0)). White circles show the states, black dots represent actions and square boxes show the terminal states. The region in red shows the part of the backup diagram considered by the respective methods for updating the estimated value function V.

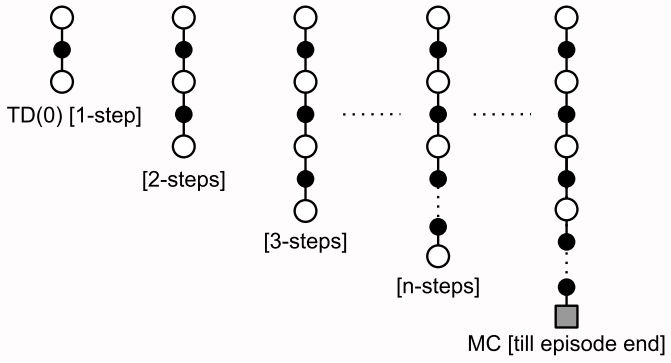
- TD can learn *before* knowing the final outcome ⇒ Less memory consumption, less peak computation
- TD can learn *without* the final outcome ⇒ Also suitable for incomplete and non-episodic tasks

	DP	MC	TD
Bootstrapping	yes	no	yes
Sampling	no	yes	yes

Table 4: Differences between DP, MC and TD

Optional Information: TD(λ)

TD(0) considers only a single step for making an update. We can also consider n steps where $n = 1, 2, 3, ...$ This leads to methods known as TD(λ). TD(λ) methods form a spectrum bewteen TD(0) where $\lambda = 0$, and MC-learning where $\lambda = 1$ (where we consider all the steps till the end of the episode).



Redrawn from [http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching_files/MC-TD.pdf]

4.2 Model-free Control

Model-free prediction enables us to evaluate how good a given policy is by estimating its value function, but on its own, it is not enough to solve an RL problem. For this, we need a way to gradually improve the policy that is being followed by the agent. Model-free control helps us to do exactly that. It refers to the problem of finding the optimal value function and policy without relying on an MDP.

On-policy vs Off-policy Learning

For the *control problem*, optimal policies can be learned in two ways:

- **On-policy Learning**: This refers to learning to improve a *target policy* π from experience sampled from the same policy. In other words, the *behavior policy* (policy used for generating samples) and the *target policy* (policy being improved) are the same.
- **Off-policy Learning**: Here, we learn about a *target policy* π from experience sampled from a different *behavior policy* μ. Here the agent learns about the optimal policy while following a non-optimal exploratory policy. Once training is complete, the agent knows the optimal policy and can utilize it.

4.2.1 SARSA

Earlier in Algo. 2, we saw that using the state-value function, the optimal policy can be computed using the following equation:

$$\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$$

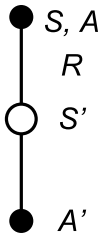
However, in order to perform the summation, we need to know the state transition probabilities given by the function p. On the other hand, if we have the action-value function Q, then we can find the best actions without needing an MDP:

$$\pi'(s) = \arg\max_a Q(s, a)$$

Given the advantages of TD over MC, it is a natural idea to apply TD for estimating the action-value function Q and then to use ϵ-greedy action selection for selecting the next action by referring to the Q-values. As in TD(0) prediction, there is no need to wait till the end of an episode. This leads us to the SARSA algorithm, which is an **on-policy TD-control algorithm**.

Algorithm 7 SARSA (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize: $Q(s, a)$, $\forall s \in \mathcal{S}^+, a \in \mathcal{A}$ arbitrarily, except $Q(terminal, \cdot) = 0$
for each episode **do**
  Initialize S
  Choose A from S using policy derived from Q (eg. ϵ-greedy)
  for each step of episode until S is terminal **do**
    Take action A, observe reward R and next state S'
    Choose A' from S' using policy derived from Q (eg. ϵ-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$ (Eq. 29)[8]
    $S \leftarrow S'; A \leftarrow A'$
  end for
end for



[8]Notice that for TD prediction (Eq. 29), the action was always chosen according to the policy π to be evaluated, but here, since we are dealing with the *control problem*, the action is chosen only by looking at Q. Gradually as $Q \approx q_*$, this will lead to the agent performing optimal actions.

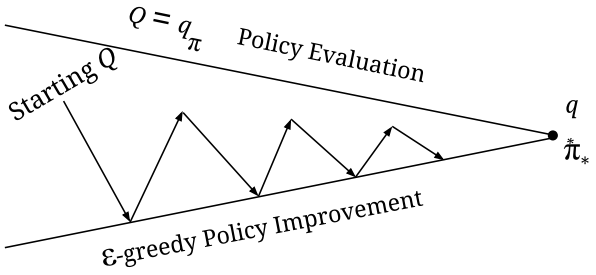


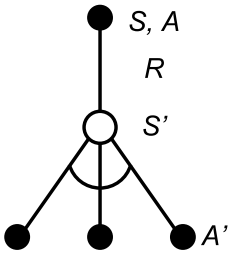
Figure 14: In SARSA, we continually estimate q_π for the behavior policy π, and at the same time change π toward greediness with respect to q_π. Redrawn from [`http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching_files/control.pdf`]

4.2.2 Q-Learning

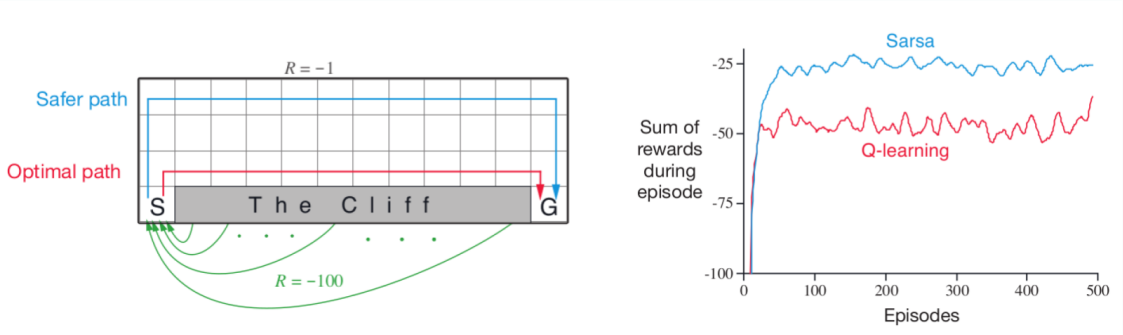
Q-learning is an **off-policy TD-control** algorithm that is used for learning the action-value function $Q \approx q_*$. Here the behavior policy μ is different from the learned target policy π. Q-learning directly approximates q_*, independent of the policy being followed. As long as all state-action pairs continue to be updated, Q-learning converges to the optimal action-value function q_*.

Algorithm 8 Q-learning (off-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize: $Q(s, a)$, $\forall s \in \mathcal{S}^+, a \in \mathcal{A}$ arbitrarily, except $Q(terminal, \cdot) = 0$
for each episode **do**
  Initialize S
  for each step of episode until S is terminal **do**
    Choose A from S using policy derived from Q (eg. ϵ-greedy)
    Take action A, observe reward R and next state S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  end for
end for



Exercise: Cliff Walking - Difference between SARSA and Q-learning (example 6.6 from [10])



Consider the gridworld and its rules, as shown on the left. This is an undiscounted episodic task with start and goal states (S and G respectively). Reward is -1 for all transitions, except when the agent steps off the cliff, for which the reward is -100. This also sends the agent to S. Actions have deterministic effects (left, right, up and down). For this problem, both SARSA and Q-learning end up finding solutions, as shown by the paths in the figure on the left.

- Why are the paths different?
- Why is the performance of Q-learning worse than SARSA (figure on the right)?
- How can we get the same solution (path to goal) from both algorithms?
- If action selection is always performed greedily, is Q-learning exactly the same algorithm as SARSA? Will they make exactly the same action selections and weight updates?

5 Value Function Approximation

In Sec. 4, we have represented the state-value function $v_\pi(s)$ and the action-value function $q_\pi(s, a)$ in the form of tables V and Q respectively. The algorithms that we have considered such as Monte Carlo prediction (Algo. 5), TD(0) learning (Algo. 6), SARSA (Algo. 7) and Q-learning (Algo. 8) directly update the entries in the V or Q tables.

Although, for problems involving a small number of states and actions, using tables to explicitly store the value of each state (or state-action pair) is feasible, for large state and action spaces, the usage of table becomes prohibitively difficult. Consider the game of Go, which has 10^{170} states. For this game, representing the value function as a table would consume a huge amount of memory. Moreover, updating the entire table would be very time consuming and expensive.

Problems in the real world often have state spaces which are much larger than games such as Go. For problems in robotics (e.g. a mobile robot moving in a room), the states are often continuous, resulting in an infinite state space. Clearly, for such problems, tabular RL methods will not suffice. For these problems, value functions are represented with the help of function approximators parameterized by some weight vector $\mathbf{w}$ (where the size of $\mathbf{w}$ is much smaller than the number of states or state-action pairs). This results in the following parameterized functions which approximate the true value functions:

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s) \tag{30}$$

$$\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a) \tag{31}$$

With this representation, we do not need to store values of individual states. It is expected that the parameterized value functions will be able to generalize sufficiently (e.g. values of neighboring states are usually similar), even to states which have not been encountered during training. The RL algorithms in this case update the parameter (weight) vector $\mathbf{w}$ during training so that the value-function ends up approximating the true value of states. It is possible to use any kind of function approximator for representing value functions, but since we need an easy way to update $\mathbf{w}$, it is common to use differentiable function approximators such as *linear combination of features* or non-linear approximators such as *neural networks*, for which $\mathbf{w}$ can be updated using gradient descent.

In this section, we extend the tabular model-free methods discussed in Sec. 4 to RL problems with continuous state spaces but discrete action spaces. Later, in Sec. 6, we will discuss how to handle continuous action spaces as well.

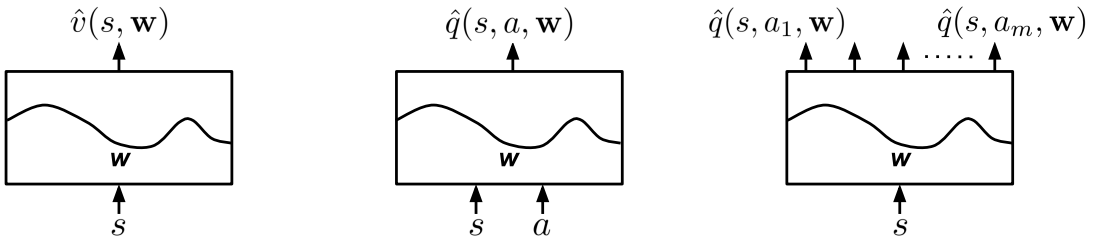


Figure 15: Value function approximation. (left) The state-value function, which outputs the value of a given state. (middle) Action-value functions can be represented as a function that takes s and a as input and gives the action-value for that pair. (right) A better way to represent q is have a function which takes the state s as input and provides the action-values for the input state and all possible actions. Here, only one forward pass is needed to calculate the action values of all actions.

5.1 Gradient Descent

Gradient descent gives us a way of adjusting the parameters $\mathbf{w}$ of the function approximator. If $J(\mathbf{w})$ is a differentiable function of the parameter vector $\mathbf{w}$, then the gradient of $J(\mathbf{w})$ is defined as:

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial w_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial w_n} \end{pmatrix} \tag{32}$$

In order to find the local minimum of $J(\mathbf{w})$, we need to adjust the parameters $\mathbf{w}$ in the direction of the negative gradient. We do this by taking a small step (determined by the learning rate or step size α) in the direction given by the negative gradient. Hence the change in $\mathbf{w}$ is given by:

$$\Delta\mathbf{w} = -\frac{1}{2}\alpha\nabla_{\mathbf{w}} J(\mathbf{w}) \tag{33}$$

Here the constant $\frac{1}{2}$ is used for mathematical convineance, as we shall see a little later. For a more detailed discussion of gradient descent, refer to [2].

5.2 Approximating Value Functions with SGD

For representing state-value functions using function approximators, the goal is to find a parameter vector $\mathbf{w}$ such that the mean-squared error between the approximate value function $\hat{v}(s, \mathbf{w})$ and the true value function $v_\pi(s)$ is minimized. Hence the loss function can be written as:

$$J(\mathbf{w}) = \mathbb{E}_\pi\Big[\big(v_\pi(s) - \hat{v}(s, \mathbf{w})\big)^2\Big] \tag{34}$$

Differentiating Eq. 34 with respect to $\mathbf{w}$, using the chain rule and noting that $v_\pi(s)$ is independent of $\mathbf{w}$, we get

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = -2\mathbb{E}_\pi\Big[\big(v_\pi(s) - \hat{v}(s, \mathbf{w})\big)\nabla_{\mathbf{w}}\hat{v}(s, \mathbf{w})\Big] \tag{35}$$

Plugging in the value of $\nabla_{\mathbf{w}} J(\mathbf{w})$ in Eq. 33, we get

$$\begin{aligned} \Delta\mathbf{w} &= -\frac{1}{2}\alpha\nabla_{\mathbf{w}} J(\mathbf{w}) \\ &= \alpha\mathbb{E}_\pi\Big[\big(v_\pi(s) - \hat{v}(s, \mathbf{w})\big)\nabla_w\hat{v}(s, \mathbf{w})\Big] \end{aligned} \tag{36}$$

The expected update in Eq. 36 gives the full gradient update for $\mathbf{w}$. **Stochastic Gradient Descent** (SGD), samples from this expectation (since SGD makes the update based on a single observation) and hence the update for SGD can simply be written as

$$\Delta\mathbf{w} = \alpha\big(v_\pi(s) - \hat{v}(s, \mathbf{w})\big)\nabla_w\hat{v}(s, \mathbf{w}) \tag{37}$$

Note that Eq. 34 is defined just the way a loss function would be written for a supervised learning problem, where true labels are available for each data point. In RL, however such true labels are not available. The problem we now have to solve is how we should obtain values for $v_\pi(s)$ so that we can utilize Eq. 37 (explained in Sec. 5.5).

Similarly, when approximating the action-value function using a function approximator $\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$, the objective function for the mean-squared error is written as

$$J(\mathbf{w}) = \mathbb{E}_\pi\Big[\big(q_\pi(s, a) - \hat{q}(s, a, \mathbf{w})\big)^2\Big] \tag{38}$$

Taking the derivative of $J(\mathbf{w})$ with respect to $\mathbf{w}$ using the chain rule, we get

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = -2\mathbb{E}_\pi\Big[\big(q_\pi(s, a) - \hat{q}(s, a, \mathbf{w})\big)\nabla_{\mathbf{w}}\hat{q}(s, a, \mathbf{w})\Big] \tag{39}$$

Similar to Eq. 36-37, stochastic gradient descent can be used to find the weight update $\Delta\mathbf{w}$ which takes a step towards the local minimum of the objective function

$$\begin{aligned}\Delta\mathbf{w} &= -\frac{1}{2}\alpha\nabla_{\mathbf{w}} J(\mathbf{w}) \\ &= \alpha\big(q_\pi(s, a) - \hat{q}(s, a, \mathbf{w})\big)\nabla_{\mathbf{w}}\hat{q}(s, a, \mathbf{w})\end{aligned} \tag{40}$$

Here too, we assume that we know the true value of $q_\pi(s, a)$. In Sec. 5.6 we will see how to get around this problem.

Weight update using SGD

For the state-value function $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$

$$\Delta\mathbf{w} = \alpha\big(v_\pi(s) - \hat{v}(s, \mathbf{w})\big)\nabla_{\mathbf{w}}\hat{v}(s, \mathbf{w})$$

For the action-value function $\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$

$$\Delta\mathbf{w} = \alpha\big(q_\pi(s, a) - \hat{q}(s, a, \mathbf{w})\big)\nabla_{\mathbf{w}}\hat{q}(s, a, \mathbf{w})$$

5.3 Linear Value Function Approximation

The simplest form of state-value function approximation is *linear value function approximation*, where a state s is represented using a feature vector $\mathbf{\Phi}$ such as

$$\mathbf{\Phi}(s) = \begin{pmatrix}\phi_1(s) \\ \phi_2(s) \\ \vdots \\ \phi_n(s)\end{pmatrix} \tag{41}$$

The approximate state-value function $\hat{v}(s, \mathbf{w})$ can then be written as a linear combination of the weight vector $\mathbf{w}$ and the feature vector $\mathbf{\Phi}$.

$$\hat{v}(s, \mathbf{w}) = \mathbf{\Phi}(s)^T\mathbf{w} = \sum_{j=1}^{n}\phi_j(s)w_j \tag{42}$$

Thus the loss function or objective in Eq. 34 can be written as

$$J(\mathbf{w}) = \mathbb{E}_\pi\Big[\big(v_\pi(s) - \hat{v}(s, \mathbf{w})\big)^2\Big]$$

$$= \mathbb{E}_\pi \Big[\big(v_\pi(s) - \mathbf{\Phi}(s)^T \mathbf{w} \big)^2 \Big] \tag{43}$$

Since $\nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w}) = \nabla_{\mathbf{w}} \mathbf{\Phi}(s)^T \mathbf{w} = \mathbf{\Phi}(s)$ (gradient of the LHS of Eq. 46), the update rule in Eq. 37 is particularly simple, and can be written as

$$\begin{aligned} \Delta \mathbf{w} &= \alpha \big(v_\pi(s) - \hat{v}(s, \mathbf{w}) \big) \nabla_w \hat{v}(s, \mathbf{w}) \\ &= \alpha \Big(v_\pi(s) - \hat{v}(s, \mathbf{w}) \Big) \mathbf{\Phi}(s) \end{aligned} \tag{44}$$

Thus, for linear value function approximation, the weight update $\Delta \mathbf{w}$ in Eq. 47 can be interpreted as the product of the step size $[\alpha]$, the prediction error $[(v_\pi(s) - \hat{v}(s, \mathbf{w}))]$ and the feature vector $[\mathbf{\Phi}(s)]$. When non-linear function approximators such as neural networks are used, the general form of the weight update in Eq. 37 remains unchanged, but in order to use it, we need to find the gradient $\nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w})$ for the neural network.

For the action-value function, a state-action pair is represented using a feature vector $\mathbf{\Phi}$.

$$\mathbf{\Phi}(s, a) = \begin{pmatrix} \phi_1(s, a) \\ \phi_2(s, a) \\ \vdots \\ \phi_n(s, a) \end{pmatrix} \tag{45}$$

and the approximate action value function $\hat{q}(s, a, \mathbf{w})$ can be written as

$$\hat{q}(s, a, \mathbf{w}) = \mathbf{\Phi}(s, a)^T \mathbf{w} = \sum_{j=1}^{n} \phi_j(s, a) w_j \tag{46}$$

Consequently, the weight update is written as

$$\begin{aligned} \Delta \mathbf{w} &= \alpha \big(q_\pi(s, a) - \hat{q}(s, a, \mathbf{w}) \big) \nabla_w \hat{q}(s, a, \mathbf{w}) \\ &= \alpha \big(q_\pi(s, a) - \hat{q}(s, a, \mathbf{w}) \big) \mathbf{\Phi}(s, a) \end{aligned} \tag{47}$$

Weight update for Linear Function Approximation

For the state-value function $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$

$$\begin{aligned} \Delta \mathbf{w} &= \alpha \big(v_\pi(s) - \hat{v}(s, \mathbf{w}) \big) \nabla_w \hat{v}(s, \mathbf{w}) \\ &= \alpha \Big(v_\pi(s) - \hat{v}(s, \mathbf{w}) \Big) \mathbf{\Phi}(s) \end{aligned}$$

For the action-value function $\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$

$$\begin{aligned} \Delta \mathbf{w} &= \alpha \big(q_\pi(s, a) - \hat{q}(s, a, \mathbf{w}) \big) \nabla_w \hat{q}(s, a, \mathbf{w}) \\ &= \alpha \big(q_\pi(s, a) - \hat{q}(s, a, \mathbf{w}) \big) \mathbf{\Phi}(s, a) \end{aligned}$$

5.4 Features for Linear Methods

Different kinds of features for linear methods can be constructed [9]

Polynomials

States of many systems are expressed as numbers (e.g. position and velocity for the pole balancing task). If a 2-dimensional state $s = [s_1, s_2]^T$ is directly used as-is with a linear method, the state representation fails to encode the iteraction between different dimensions of the state, which may be useful for speeding the learning process. Instead we can use a 4-dimensional feature vector $\mathbf{\Phi}(s) = [1, s_1, s_2, s_1 s_2]^T$ which captures the interaction between s_1 and s_2. In general, if state $\mathbf{s}$ has k dimensions, $s_1, s_2, ..., s_k$ with each $s_i \in \mathbb{R}$, each order-n polynomial-basis feature can be written as

$$\phi_i(\mathbf{s}) = \prod_{j=1}^{k} s_j^{c_{i,j}} \tag{48}$$

where each $c_{i,j}$ is an integer in the set $\{0, 1, ..., n\}$ for $n \geq 0$. For a state with k dimensions these $\phi_i(s)$ features make up the feature vector $\mathbf{\Phi}$ which is has $(n+1)^k$ different features.

Coarse Coding

For tasks, in which the state state set is a continuous two dimenisonal place, any position can be represented using features corresponding to circles in state space, as shown in Fig. 16 (left). Each circle represents a binary feature which is 1 if the position is inside the circle and 0 otherwise. The features for all the circles together form the feature vector $\mathbf{\Phi}(s)$ for a given state s.

Tile Coding

In Tile Coding 16 (right), the state space is divided into partitions. Each partition is called a *tiling* and each element (grid cell) of a tiling is called a *tile*. Different tilings are partly offset from each other. For a particular position in the state space, only the features for those tiles which contain the position will be 1 and all other features will be 0, leading to a binary feature vector (which has one feature for each tile in each tiling).

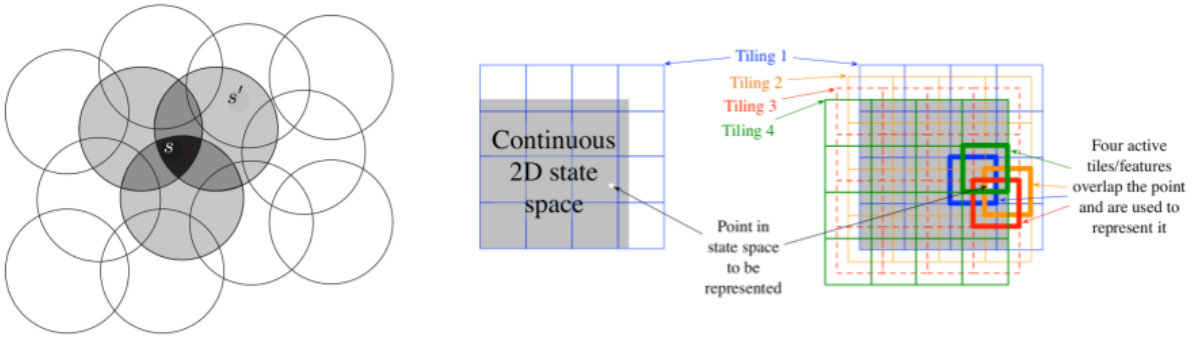


Figure 16: (left) Coarse coding (right) Tile coding

[9] For further details see [10] pg. 210.

Radial Basis Functions

Radial Basis Functions (RBFs) can be considered a generalization of coarse coding to the continuous domain, where instead of binary features, the features contain real numbers in the range [0,1]. Each feature ϕ_i is a Gaussian, dependent on the state s, the feature's center c_i, and the feature's width σ_i.

$$\phi_i(s) = exp\Big(- \frac{||s - c_i||^2}{2\sigma_i^2} \Big) \tag{49}$$

Exercise: Polynomial Features

Recall:
$\phi_i(\mathbf{s}) = \prod_{j=1}^{k} s_j^{c_{i,j}}$ where $\mathbf{s} = [s_1, s_2, ..., s_k]^T$, $c_{i,j} \in \{0, 1, ..., n\}$
for $n \geq 0$. For k dimensions, $\mathbf{\Phi}$ has $(n+1)^k$ different features.

Let $\mathbf{s} = [s_1, s_2]^T$, $(k = 2)$ and let $n = 2$. What is the polynomial feature vector $\mathbf{\Phi}$?

5.5 Algorithms for Prediction

While formulating Eq. 34-47, we have assumed that the true state-value $v_\pi(s)$ is known. However, in RL, there is no supervisor and so there is no direct way to know this value. In order to circumvent this issue, we substitute a target for $v_\pi(s)$ instead of using the true value. We do this by using either the Monte Carlo target (Eq. 26) or the Temporal Difference target (Eq. 28).

Weight Update for MC and TD prediction

For MC, the target us the return G_t. Thus the weight update is

$$\Delta \mathbf{w} = \alpha \Big(G_t - \hat{v}(S_t, \mathbf{w}) \Big) \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}) \tag{50}$$

For TD(0), the target is the TD target $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$. Hence,

$$\Delta \mathbf{w} = \alpha \Big(R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}) \Big) \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}) \tag{51}$$

We can directly use Eq. 50 to modify Algo. 5 to obtain the gradient MC algorithm for estimating $\hat{v} \approx v_\pi$.

Algorithm 9 Gradient MC Prediction, for estimating $\hat{v} \approx v_\pi$

Input:
  Policy π to be evaluated
  Algorithm parameter: step size $\alpha > 0$
  A differentiable value function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
Initialize
  Value-function weight vector $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g. $\mathbf{w} = 0$)
repeat
  Generate an episode using π: $S_0, A_0, R_1, S_1, A_1, R_2, ..., S_{T-1}, A_{T-1}, R_T$
  for each step of episode, $t = 0, 1, ..., T-1$ **do**
    $\mathbf{w} = \mathbf{w} + \alpha\Big(G_t - \hat{v}(S_t, \mathbf{w})\Big)\nabla_{\mathbf{w}}\hat{v}(S_t, \mathbf{w})$
  end for
until forever

Since the MC update uses the actual return G_t (which is an unbiased estimate of $v_\pi(S_t)$, gradient-descent version of MC prediction is guaranteed to converge to a locally optimal solution ([10] pg. 202).

Similarly, we can modify the algorithm for tabular TD(0) (Algo. 6) by using function approximation as shown in Algo. 10.

Algorithm 10 Semi-gradient TD(0) Prediction, for estimating $\hat{v} \approx v_\pi$

Input:
  Policy π to be evaluated
  Algorithm parameter: step size $\alpha > 0$
  A differentiable value function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(terminal) = 0$
Initialize
  Value-function weight vector $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g. $\mathbf{w} = [0, ..., 0]^T$)
repeat
  Initialize S
  for each step of episode until S is terminal **do**
    Choose $A \sim \pi(\cdot|S)$
    Take action A, observe R, S'
    $\mathbf{w} = \mathbf{w} + \alpha\Big(R + \gamma\hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})\Big)\nabla_{\mathbf{w}}\hat{v}(S, \mathbf{w})$
    $S \leftarrow S'$
  end for
until forever

However, for the gradient-descent version of TD prediction, the situation is a bit more complicated. Recall that while deriving Eq. 35, we noted that the target v_π was independent of the weight vector $\mathbf{w}$. But for TD, we bootstrap using the TD target $R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w})$ (Eq. 51) which depends on the current value of $\mathbf{w}$.

This implies that the TD target is a biased estimate of the true return and will not result in a true gradient-descent method. For TD, we take ignore the effect of $\mathbf{w}$ on the target. This results in bootstrapping methods such as TD including only a part of the true gradient and accordingly, they are called *semi-gradient methods*. Although, bootstrapping

methods offer less robust convergence guarantees, they still offer advantages such as being applicable for both continuous and episodic tasks and enabling faster learning. Hence in many instances, TD methods, such as Algo. 10 are preferable.

5.6 Algorithms for Control

For the control problem using function approximation, we use the action-value function and follow the same general process as tabular methods for control. The difference is that for function approximation, the weights $\mathbf{w}$ are updated instead of entries in the table, as shown in Fig. 17.

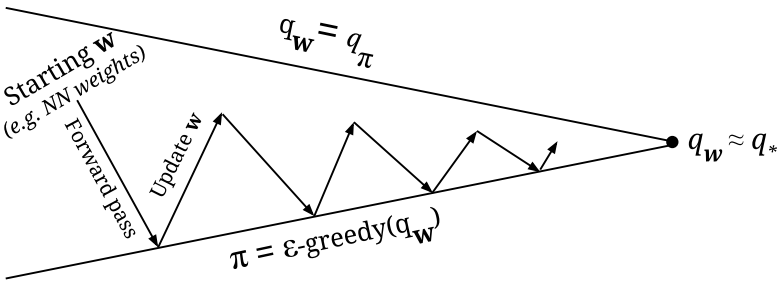


Figure 17: Function approximation for control. The policy evaluation step (top) uses a parameterized representation of the action value function $\hat{q}(\cdot, \cdot, \mathbf{w}) \approx q_\pi$. The policy improvement step (bottom) uses an ϵ-greedy action selection to improve the policy implied by the current action-value function.

Weight Update for TD control

For TD(0), the target is the TD target $R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w})$. Hence,

$$\Delta \mathbf{w} = \alpha \Big(R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}) \Big) \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w}) \tag{52}$$

Using Eq. 52, we get the *semi-gradient* SARSA algorithm shown in Algo. 11 (by modifying the SARSA algorithm in Algo. 7).

Algorithm 11 Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: A differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize value function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g. $\mathbf{w} = [0, ..., 0]^T$)
for each episode **do**
  Initialize S
  Choose A from S using policy derived from $\hat{q}$ (eg. ϵ-greedy)
  for each step of episode **do**
    Take action A, observe reward R and next state S'
    if S' is terminal **then**
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big(R - \hat{q}(S, A, \mathbf{w})\big)\nabla_{\mathbf{w}}\hat{q}(S, A, \mathbf{w})$
      Go to next episode
    end if
    Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (eg. ϵ-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big(R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big)\nabla_{\mathbf{w}}\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'; A \leftarrow A'$
  end for
end for

5.7 Convergence Properties and the Deadly Triad

When function approximation is used, we typically have to settle for a local optimum. The limited expressivity of the parametric value function restricts the scope of learnable value functions and thus policies. Due to $\hat{q}$ being approximate, convergence guarantees are weaker than in the case of tabular methods. Particularly, the risk of divergence of the learned value function arises if we combine three things known as the ***Deadly Triad***:

- **Function Approximation** - Using a parametric function to significantly generalize from a large number of examples.
- **Bootstrapping** - Learning value estimates from other value estimates.
- **Off-policy Learning** - Learning about a policy from data not due to that policy.

The following table shows the convergence properties of some control algorithms using different representations of the value functions.

Algorithm	***Tabular***	***Linear Func. Approx.***	***Non-linear Func. Approx.***
SARSA	*Yes*	*Chatters around optimal*	*No*
Q-Learning	*Yes*	*No*	*No*

6 Policy Gradients

In Sec. 5, we have approximated the state-value or the action-value value function using parameters $\mathbf{w}$

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$
$$\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$$

For solving the *control* problem, we needed a policy which was generated directly from the value function (e.g. using ϵ-greedy action selection).

Instead of using an implicit policy, now we will directly parameterize the policy itself using parameters $\boldsymbol{\theta}$, and then find a way to change $\boldsymbol{\theta}$ so that we can compute the best possible policy. The parameterized policy is represented as

$$\pi_{\boldsymbol{\theta}}(s, a) = \mathbb{P}\big[a|s, \boldsymbol{\theta}\big] \tag{53}$$

We will look at methods which optimize $\boldsymbol{\theta}$ directly without needing to go through value-functions.

6.1 Advantages of Policy-based RL

Policy-based RL methods have some advantages when compared to value-based methods:

- **Better Convergence**: For a policy approximated with a continuous function approximator, making small changes to the policy parameters $\boldsymbol{\theta}$ will result in small changes in the actions to be taken in a given state. This results in better convergence properties than for value function approximation where a small change in the parameters $\mathbf{w}$ can result in a large change in the action to be taken (using ϵ-greedy action selection).
- **Continuous action spaces**: For the value-based methods, the action is chosen by looking at the action-values of different actions in a given state and then choosing (greedily or ϵ-greedily) the action which has the highest value. This is only possible if actions are discrete and finite. When the policy is directly parameterized, it is possible to generate continuous actions, and thus policy-based methods may be applicable to a greater range of problems.
- **Simpler to learn Policy directly**: For some problems, estimating the value function can be a harder problem than directly learning the policy. For example, when an RL agent learns to play the game of Pong[10] directly by looking at pixels on the screen, it may be more difficult to estimate the future score (return) for a particular state (screen frame) than to figure out whether the paddle should move left or right based on the position of the ball.

6.2 Policy Optimization

For a given parameterized policy $\pi_{\boldsymbol{\theta}}$, our goal is to find the best possible $\boldsymbol{\theta}$ so that the agent can collect the maximum amount of rewards. In order to find the best policy, we need a way to determine the quality of the policy $\pi_{\boldsymbol{\theta}}$. This is given by the objective function.

[10] `https://en.wikipedia.org/wiki/Pong`

Policy Objective Function for Episodic Tasks

For episodic tasks, we can use the value of the start state s_0, which gives the expected return when starting in s_0 and thereafter following policy $\pi_{\boldsymbol{\theta}}$

$$J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(s_0) \tag{54}$$

Policy-based RL is an optimization problem, where our task is to find a $\boldsymbol{\theta}$ that maximizes the objective function $J(\boldsymbol{\theta})$. Policy gradient algorithms search for a local maximum in $J(\boldsymbol{\theta})$ by ascending the gradient of the policy with respect to the parameters $\boldsymbol{\theta}$. The change in $\boldsymbol{\theta}$ is obtained by taking a small step (determined by the step size α) in the direction of the gradient of the objective function as shown below:

$$\Delta\boldsymbol{\theta} = \alpha\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{55}$$

Here, the gradient of the objective function is given by

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_n} \end{pmatrix} \tag{56}$$

In Sec. 5, the objective function we used was the squared error between the target value and the actual value (Eq. 34 and 38) and so we performed gradient descent to minimize this error. Here, our goal is to maximize performance and so we do gradient ascent instead.

6.3 Policy Gradient with Finite Differences

The simplest way of estimating the policy gradient is by using finite differences [6]. A given n-dimensional policy parameter vector $\boldsymbol{\theta}_h$ is varied n times (once in each dimension) to produce increments $\Delta\boldsymbol{\theta}_i, i = 1, .., n$. For each of these increments, a policy variation $J(\boldsymbol{\theta}_h + \Delta\boldsymbol{\theta}_i)$ is produced. Rollouts are performed for estimating $\Delta\hat{J}_i = J(\boldsymbol{\theta}_h + \Delta\boldsymbol{\theta}_i) - J(\boldsymbol{\theta}_h)$. This results in the data points $[\Delta\boldsymbol{\theta}_1, \Delta\hat{J}_1], ..., [\Delta\boldsymbol{\theta}_n, \Delta\hat{J}_n]$ over which regression can be performed to compute an estimate of the policy gradient.

Algorithm 12 Policy Gradient estimation using Finite Differences

Input: Policy parameterization instance $\boldsymbol{\theta}_h$ (n-dimensional vector)
for $i = 1$ to n **do**
  Generate policy variation $\Delta\boldsymbol{\theta}_i$ by perturbing $\boldsymbol{\theta}_h$ in dimension i
  Estimate $\hat{J}_i \approx J(\boldsymbol{\theta}_h + \Delta\boldsymbol{\theta}_i) = \sum_{k=0}^{H} \gamma^k R_k$ from roll-out
  Estimate $\hat{J}_{ref} = J(\boldsymbol{\theta}_h)$ from roll-out
  Compute $\Delta\hat{J}_i \approx J(\boldsymbol{\theta}_h + \Delta\boldsymbol{\theta}_i) - \hat{J}_{ref}$
end for
Return the policy gradient estimate
$\mathbf{g}_{FD} \approx \nabla_{\boldsymbol{\theta}} J|_{\boldsymbol{\theta}=\boldsymbol{\theta}_h} = \left(\Delta\boldsymbol{\theta}^T\Delta\boldsymbol{\theta}\right)^{-1}\Delta\boldsymbol{\theta}^T\Delta\hat{\mathbf{J}}$
where $\Delta\boldsymbol{\theta} = [\Delta\boldsymbol{\theta}_1, ..., \Delta\boldsymbol{\theta}_n]^T$ and $\Delta\hat{\mathbf{J}} = [\Delta\hat{J}_1, ..., \Delta\hat{J}_n]^T$
(performing regression over the data points $[\Delta\boldsymbol{\theta}_1, \Delta\hat{J}_1], ..., [\Delta\boldsymbol{\theta}_n, \Delta\hat{J}_n]$)

Using finite differences to compute the policy gradient is a naive numerical approach and it can be noisy and inefficient, mainly because we need to perform rollouts for each dimension of the policy parameter vector. For a high-dimensional parameterization like a neural network, this is simply not possible. However, in some situations (such as [3]) this approach can be effective.

6.4 Score Function and Liklihood Ratio

In order to compute the policy gradient analytically, we assume that the policy $\pi_{\boldsymbol{\theta}}$ is differentiable whenever it is non-zero (whenever actions are picked) and that we know the gradient of our policy $\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(s,a)$ (since we construct the policy function such as a *softmax* policy or a *neural network* ourselves and we can derive the gradient for this function). *Liklihood Ratios* explore the following identity:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(s,a) &= \pi_{\boldsymbol{\theta}}(s,a)\frac{\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(s,a)}{\pi_{\boldsymbol{\theta}}(s,a)} \quad \text{(Mutipying and dividing by } \pi_{\boldsymbol{\theta}}(s,a)\text{)} \\ &= \pi_{\boldsymbol{\theta}}(s,a)\nabla_{\boldsymbol{\theta}}\ln\pi_{\boldsymbol{\theta}}(s,a) \quad \text{(since } \nabla_{\boldsymbol{\theta}}\ln(z) = \frac{1}{z}\nabla_{\boldsymbol{\theta}}z\text{)} \end{aligned} \tag{57}$$

The term $\nabla_{\boldsymbol{\theta}}\ln\pi_{\boldsymbol{\theta}}(s,a)$ is known as the *score function* which tells us how to adjust our policy so that the liklihood of choosing good actions is increased.

Score Function for Policies with Discrete Actions

For discrete actions, we can use a linear combination of features for representing the weightage of each action, such as $\Phi(s,a)^T\boldsymbol{\theta} \in \mathbb{R}$, where $\Phi(s,a)$ is the feature vector and $\boldsymbol{\theta}$ is the parameter vector of our policy. The weightage of the actions can then be expressed as probabilities by using the *softmax function*. Hence, our policy will be

$$\pi_{\boldsymbol{\theta}}(s,a) = \frac{e^{\boldsymbol{\Phi}(s,a)^T\boldsymbol{\theta}}}{\sum_b e^{\boldsymbol{\Phi}(s,b)^T\boldsymbol{\theta}}} \tag{58}$$

where the denominator in the RHS sums over all actions and normalizes the weights for actions into probabilities $\in [0,1]$. For this case, the score function is

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} \ln \pi_{\boldsymbol{\theta}}(s,a) &= \nabla_{\boldsymbol{\theta}} \ln \frac{e^{\Phi(s,a)^T\boldsymbol{\theta}}}{\sum_b e^{\Phi(s,b)^T\boldsymbol{\theta}}} \\
&= \nabla_{\boldsymbol{\theta}} \Big(\ln e^{\Phi(s,a)^T\boldsymbol{\theta}} - \ln \sum_b e^{\Phi(s,b)^T\boldsymbol{\theta}} \Big) \\
&= \nabla_{\boldsymbol{\theta}} \ln e^{\Phi(s,a)^T\boldsymbol{\theta}} - \nabla_{\boldsymbol{\theta}} \ln \sum_b e^{\Phi(s,b)^T\boldsymbol{\theta}} \\
&= \nabla_{\boldsymbol{\theta}} \Phi(s,a)^T\boldsymbol{\theta} - \nabla_{\boldsymbol{\theta}} \ln \sum_b e^{\Phi(s,b)^T\boldsymbol{\theta}} \\
&= \Phi(s,a) - \nabla_{\boldsymbol{\theta}} \ln \sum_b e^{\Phi(s,b)^T\boldsymbol{\theta}} \\
&= \Phi(s,a) - \Big(\frac{1}{\sum_b e^{\Phi(s,b)^T\boldsymbol{\theta}}} \nabla_{\boldsymbol{\theta}} \sum_b e^{\Phi(s,b)^T\boldsymbol{\theta}} \Big) \quad \text{(using chain rule)} \\
&= \Phi(s,a) - \Big(\frac{1}{\sum_b e^{\Phi(s,b)^T\boldsymbol{\theta}}} \sum_b \nabla_{\boldsymbol{\theta}} e^{\Phi(s,b)^T\boldsymbol{\theta}} \Big) \quad \text{(taking gradient inside the sum)} \\
&= \Phi(s,a) - \Big(\frac{1}{\sum_b e^{\Phi(s,b)^T\boldsymbol{\theta}}} \sum_b e^{\Phi(s,b)^T\boldsymbol{\theta}} \nabla_{\boldsymbol{\theta}} \Phi(s,b)^T\boldsymbol{\theta} \Big) \quad \text{(chain rule again)} \\
&= \Phi(s,a) - \Big(\frac{1}{\sum_b e^{\Phi(s,b)^T\boldsymbol{\theta}}} \sum_b e^{\Phi(s,b)^T\boldsymbol{\theta}} \Phi(s,b) \Big) \\
&= \Phi(s,a) - \Big(\sum_b \frac{e^{\Phi(s,b)^T\boldsymbol{\theta}}}{\sum_b e^{\Phi(s,b)^T\boldsymbol{\theta}}} \Phi(s,b) \Big) \quad \text{(rearranging the summations)} \\
&= \Phi(s,a) - \Big(\sum_b \pi_{\boldsymbol{\theta}}(s,b)\Phi(s,b) \Big) \quad \text{(using definition of } \pi_{\boldsymbol{\theta}}(s,b)) \\
&= \Phi(s,a) - \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \Big[\Phi(s,\cdot) \Big] \quad \text{(since } \pi_{\boldsymbol{\theta}}(s,b) \text{ is the probability of action } b)
\end{aligned} \tag{59}$$

Note: This derivation is based on [9].

Score Function for Policies with Continuous Actions

When we want our parameterized policy function to output continuous actions instead of discrete actions, we can represent the policy function using a *Gaussian* function $\mathcal{N}(\mu(s,\boldsymbol{\theta}),\sigma^2)$, where the mean μ is a function parameterized by $\boldsymbol{\theta}$ (obtained by a linear combination of state features $\mu(s,\boldsymbol{\theta}) = \Phi(s)^T\boldsymbol{\theta}$) and the variance σ is a constant (for a more complicated case, σ can also be a parameterized function). Continuous actions $a \in \mathbb{R}$ can then sampled from this Gaussian distribution $a \sim \mathcal{N}$. Therefore, the policy function (probability of choosing a continuous action a in state s) can be written as:

$$\pi_{\boldsymbol{\theta}}(s,a) = \frac{1}{\sigma\sqrt{2\pi}} exp\Big(- \frac{\big(a-\mu(s,\boldsymbol{\theta})\big)^2}{2\sigma^2}\Big) \tag{60}$$

The score function for this Gaussian policy is

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} \ln \pi_{\boldsymbol{\theta}}(s,a) &= \nabla_{\boldsymbol{\theta}} \ln \frac{1}{\sigma\sqrt{2\pi}} exp\Big(- \frac{\big(a-\mu(s,\boldsymbol{\theta})\big)^2}{2\sigma^2}\Big) \\
&= \nabla_{\boldsymbol{\theta}} \ln exp\Big(- \frac{\big(a-\mu(s,\boldsymbol{\theta})\big)^2}{2\sigma^2}\Big) - \nabla_{\boldsymbol{\theta}} \ln \sigma\sqrt{2\pi} \\
&= \nabla_{\boldsymbol{\theta}} \Big(- \frac{\big(a-\mu(s,\boldsymbol{\theta})\big)^2}{2\sigma^2}\Big) \\
&= -\frac{1}{2\sigma^2} \cdot 2\big(a-\mu(s,\boldsymbol{\theta})\big) \cdot -\nabla_{\boldsymbol{\theta}}\mu(s,\boldsymbol{\theta}) \quad \text{(Using the chain rule)} \\
&= \frac{\big(a-\Phi(s)^T\boldsymbol{\theta}\big)\Phi(s)}{\sigma^2} \quad \text{(Since } \mu(s,\boldsymbol{\theta}) = \Phi(s)^T\boldsymbol{\theta}\text{)}
\end{aligned} \tag{61}$$

6.5 Policy Gradient Theorem

The Policy Gradient Theorem expresses the policy gradient in terms of the score function $\ln \pi_{\boldsymbol{\theta}}(s,a)$. It states that for any differentiable policy $\pi_{\boldsymbol{\theta}}(s,a)$, the policy gradient is

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\big[\nabla_{\boldsymbol{\theta}} \ln \pi_{\boldsymbol{\theta}}(s,a) \cdot q_{\pi_{\boldsymbol{\theta}}}(s,a)\big] \tag{62}$$

The Monte Carlo Policy Gradient algorithm (REINFORCE) samples episode trajectories and updates the policy function parameters by stochastic gradient ascent (thereby eliminiating the expectation). It uses the return G_t as an unbiased estimate of $q_{\pi_{\boldsymbol{\theta}}}(s,a)$.

Algorithm 13 Monte Carlo Policy Gradient Control (REINFORCE) for π_*

Input: A differentiable policy parameterization $\pi_{\boldsymbol{\theta}}(s, a)$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^d$ (e.g. to 0)
for each episode **do**
  Generate an episode $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$ by following $\pi_{\boldsymbol{\theta}}(\cdot, \cdot)$
  for each step of episode $t = 0, 1, ..., T-1$ **do**
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} \ln \pi_{\boldsymbol{\theta}}(S_t, A_t) \cdot \gamma^T G$
  end for
end for

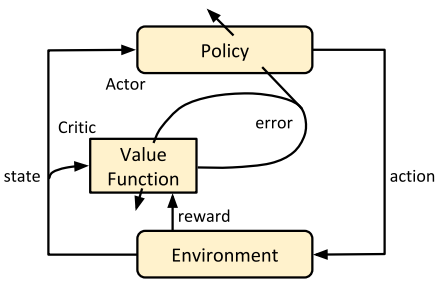
If we use linear combination of features as the parameterization of our poicy function and use either a softmax policy or a Gaussian policy, we can directly use Eq. 59 (softmax) or Eq. 61 (Gaussian) to substitute $\nabla_{\boldsymbol{\theta}} \ln \pi_{\boldsymbol{\theta}}(S_t, A_t)$ in Algo. 13.

6.6 Actor-Critic Methods

Monte Carlo Policy Gradient (REINFORCE) using only $\pi_{\boldsymbol{\theta}}(a|s)$ has a high variance. Actor-Critic methods use the value function as a **critic** to reduce this variance.

Algorithm 14 One-step Actor–Critic

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s, \mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0, \alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to 0)
for each episode **do**
  Initialize S (first state of episode)
  $I \leftarrow 1$
  while S is not terminal **do**
    $A \sim \pi(\cdot|s, \boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$ (if S' is terminal, then $\hat{v}(S', \mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S, \mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A|S, \boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$
  end while
end for



REINFORCE with Baseline vs Actor-Critic

- What is the difference between using REINFORCE with Baseline and Actor-Critic?
- What is the effect of having this difference?

Update Equations for REINFORCE with Baseline

$\delta \leftarrow G - \hat{v}(S_t, \mathbf{w})$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w})$

$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \nabla_{\boldsymbol{\theta}} \log \pi(A_t | S_t, \boldsymbol{\theta})$

Update Equations for Actor-Critic

$\delta \leftarrow R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S, \mathbf{w})$

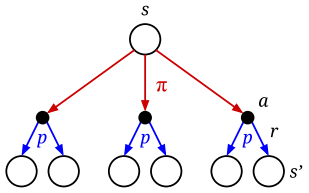
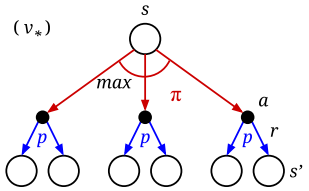
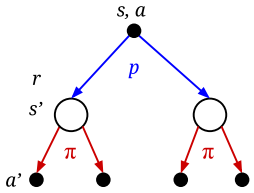
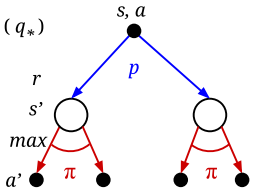
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A | S, \boldsymbol{\theta})$

7 Summary

Markov Decision Process

Term	Description	Expression
MDP	Framework defining agent-environment interaction	$\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, r, \gamma \rangle$ where • $\mathcal{S}$ is a finite set of states. • $\mathcal{A}$ is a finite set of actions. • $\mathcal{P}$ is a state transition prob. func. $\mathcal{P}(s'\|s,a) = \mathbb{P}[S_{t+1} = s'\|S_t = s, A_t = a]$ • r is a reward function $r(s,a,s') = \mathbb{E}[R_{t+1}\|S_t = s, A_t = a, S_{t+1} = s']$ • γ is a discount factor, $0 \leq \gamma \leq 1$
Markov Property	Current state includes all information about the past	-
Reward	Scalar quantity for evaluating the agent's action.	$R_t \in \mathbb{R}$
Return	Discounted sum of future rewards.	$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ $= R_{t+1} + \gamma G_{t+1}$
Goal	Maximize expected Return	$maximize(\mathbb{E}[G_t])$ at each t
Policy	Mapping from states to probabilities of actions.	$\pi(a\|s) = \mathbb{P}[A_t = a\|S_t = s]$
State-value Function	Expected **Return** when starting in s and following π thereafter.	$v_\pi(s) = \mathbb{E}_\pi[G_t\|S_t = s]$ $= \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\|S_t = s]\ \forall s \in \mathcal{S}$
Action-value Function	Expected **Return** when starting in s, taking action a and following π thereafter.	$q_\pi(s,a) = \mathbb{E}_\pi[G_t\|S_t = s, A_t = a]$ $= \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\|S_t = s, A_t = a]$ $\forall s \in \mathcal{S}, a \in \mathcal{A}$

Bellman Equations

Name	Backup Diagram	Equation
Bellman Expect. Eq. for v_π		$v_\pi(s) = \mathbb{E}_\pi\left[R_{t+1} + \gamma v_\pi(s') \mid S_t = s\right]$ $= \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_\pi(s')\right]$
Bellman Optimality Eq. for v_*		$v_*(s) = \max_a \mathbb{E}_{\pi_*}\left[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a\right]$ $= \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_*(s')\right]$
Bellman Expect. Eq. for q_π		$q_\pi(s,a) = \mathbb{E}_\pi\left[R_{t+1} + \gamma q_\pi(s', a') \mid S_t = s, A_t = a\right]$ $= \sum_{s'} p(s' \mid s, a)\left[r + \gamma \sum_{a'} \pi(a'\|s') q_\pi(s', a')\right]$
Bellman Optimality Eq. for q_*		$q_*(s,a) = \mathbb{E}_{\pi_*}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a\right]$ $= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right]$

Dynamic Programming

Prediction: How good is a policy?

- **Input** MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, r, \gamma \rangle$ and the policy π
- **Output** Value function v_π

Control: What is the best policy?

- **Input** MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, r, \gamma \rangle$
- **Output** Optimal value function v_* and the optimal policy π_*

DP Method	Algorithm	Equation
Policy Evaluation	Algo. 1	$V(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$
Policy Iteration	Algo. 2	Policy Evaluation + $\pi(s) = \arg\max_a \sum_{s',r} p(s', r\|s, a)[r + \gamma V(s')]$
Value Iteration	Algo. 3	$V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$

Model-free Prediction

MF Method	Algorithm	Equation
Monte Carlo	Algo. 5	$V(S_t) \leftarrow V(S_t) + \alpha\big(G_t - V(S_t)\big)$
Temporal Difference (0)	Algo. 6	$V(S) \leftarrow V(S) + \alpha\big(R + \gamma V(S') - V(S)\big)$

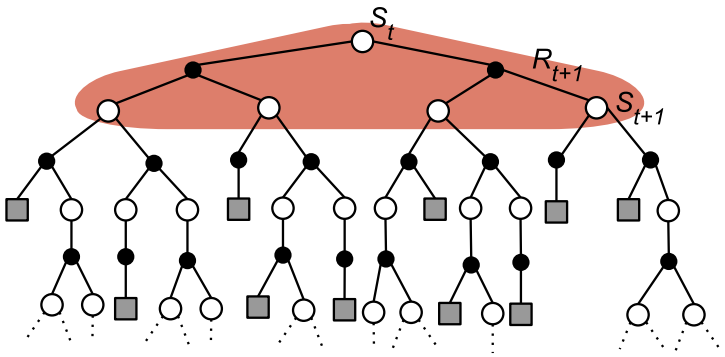


Figure 18: **DP**: Updates use $\mathbb{E}$.

$$V(s) \leftarrow \sum_a \pi\big(a \mid s\big) \sum_{s',r} p\big(s', r \mid s, a\big)\big[r + \gamma V(s')\big]$$

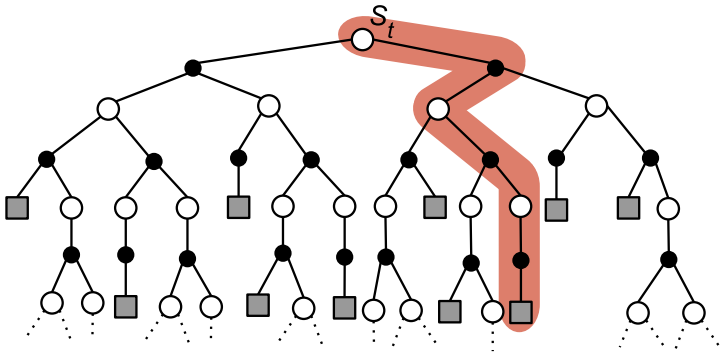


Figure 19: **MC**: Updates use true return.

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

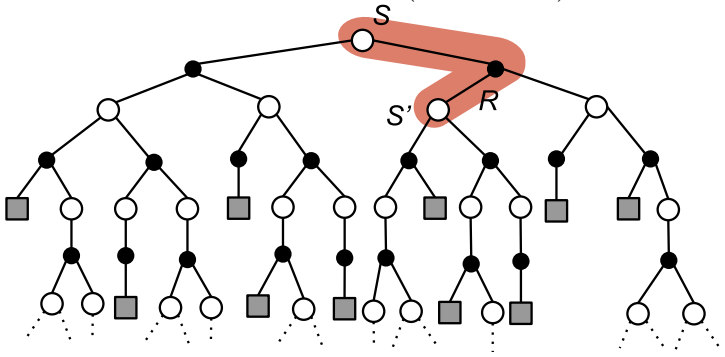
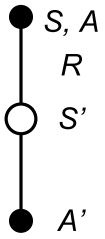


Figure 20: **TD(0)**: Updates are done after a single step.

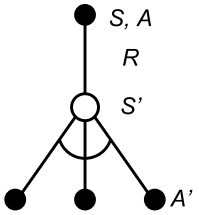
$$V(S) \leftarrow V(S) + \alpha\big(R + \gamma V(S') - V(S)\big)$$

Model-free Control

MF Method	Algorithm	Equation
SARSA	Algo. 7	$Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
Q-Learning	Algo. 8	$Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$



SARSA



Q-Learning

Value Function Approximation

VFA Method	Algorithm	Equation
Gradient MC Prediction	Algo. 9	$\Delta\mathbf{w} = \alpha\Big(G_t - \hat{v}(S_t,\mathbf{w})\Big)\nabla_{\mathbf{w}}\hat{v}(S_t,\mathbf{w})$
Semi-gradient TD(0) Prediction	Algo. 10	$\Delta\mathbf{w} = \alpha\Big(R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w})\Big)\nabla_{\mathbf{w}}\hat{v}(S_t,\mathbf{w})$
Semi-gradient SARSA	Algo. 11	$\Delta\mathbf{w} = \alpha\Big(R_{t+1} + \gamma\hat{q}(S_{t+1},A_{t+1},\mathbf{w}) - \hat{q}(S_t,A_t,\mathbf{w})\Big)\nabla_{\mathbf{w}}\hat{q}(S_t,A_t,\mathbf{w})$

Policy Gradients

PG Method	Algorithm	Equation
REINFORCE (MC PG Control)	Algo. 13	$\Delta\boldsymbol{\theta} = \alpha\nabla_{\boldsymbol{\theta}} \ln \pi_{\boldsymbol{\theta}}(S_t,A_t) \cdot \gamma^T G$

8 Learning Resources

(To be updated)

9 References

[1] Pieter Abbeel, Adam Coates, and Andrew Y Ng. Autonomous helicopter aerobatics through apprenticeship learning. *The International Journal of Robotics Research*, 29(13):1608–1639, 2010.

[2] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[3] Nate Kohl and Peter Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *Proceedings of the IEEE International Conference on Robotics and Automation*, May 2004.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[5] OpenAI, :, Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. Learning dexterous in-hand manipulation, 2018. `arXiv:1808.00177`.

[6] J. Peters. Policy gradient methods. *Scholarpedia*, 5(11):3698, 2010. revision #137199. `doi:10.4249/scholarpedia.3698`.

[7] J. Peters, J. Kober, K. Muelling, D. Nguyen-Tuong, and O. Kroemer. Towards motor skill learning for robotics. In *Proceedings of the International Symposium on Robotics Research (ISRR), Invited Paper*, 2009. URL: `http://www-clmc.usc.edu/publications/P/Peters_ISRR2009.pdf`.

[8] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[9] David Silver and Gerald Tesauro. Monte-carlo simulation balancing. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 945–952. ACM, 2009.

[10] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning*. The MIT Press, 2nd edition, 2018. URL: `https://mitpress.mit.edu/books/reinforcement-learning-second-edition`.