Lecture Notes:

Continuous Markov Decision Process and Policy Search

Course: Reinforcement learning, Leiden University

Written by: Thomas Moerland

Contents

Part I
Mathematical preliminaries

This section quickly recaps some of the mathematical notation of: 1) sets & functions, and 2) probability distributions. You should have seen these in previous courses, but we will need them throughout the lecture notes, and to understand RL in general. We will also discuss how to differentiate through an expectation, which frequently appears throughout machine learning.

1 Sets and functions

1.1 Sets

Discrete set A set of countable elements.

Examples:

- $\mathcal{X} = \{1, 2, .., n\}$ (integers)
- $\mathcal{X} = \{\text{up,down,left,right}\}$ (arbitrary elements)
- $\mathcal{X} = \{0, 1\}^d$ (d-dimensional binary space)

Continuous set A set of connected elements.

Examples:

- $\mathcal{X} = [2, 11]$ (bounded interval)
- $\mathcal{X} = \mathbb{R}$ (real line)
- $\mathcal{X} = [0, 1]^d$ (d-dimensional hypercube)

Conditioning a set We can also condition within a set, by using : or |. For example, the *discrete probability k-simplex*, which is what we actually use to define a discrete probability distribution over k categories, is given by:

$$\mathcal{X} = \{x \in [0, 1]^k : \sum_k x_k = 1\}.$$

This means that x is a vector of length k, consisting of entries between 0 and 1, with the restriction that the vector sums to 1.

Cardinality and dimensionality It is important to distinguish the cardinality and dimensionality of a set:

- The *cardinality* (size) counts the number of elements in a vector space, for which we write $|\mathcal{X}|$.

- The *dimensionality* counts the number of dimensions in the vector space $\mathcal{X}$, for which we write $\text{Dim}(\mathcal{X})$.

Examples:

- The discrete space $\mathcal{X} = \{0, 1, 2\}$ has cardinality $|\mathcal{X}| = 3$ and dimensionality $\text{Dim}(\mathcal{X}) = 1$.
- The discrete vector space $\mathcal{X} = \{0, 1\}^4$ has cardinality $|\mathcal{X}| = 2^4 = 16$ and dimensionality $\text{Dim}(\mathcal{X}) = 4$.

Cartesian product We can combine two space by taking the Cartesian product, denoted by $\times$, which consist of all the possible combinations of elements in the first and second set:

$$\mathcal{X} \times \mathcal{Z} = \{(x, z) : x \in \mathcal{X}, z \in \mathcal{Z}\}$$

We can also combine discrete and continuous spaces through Cartesian products.

Example: Assume $\mathcal{X} = \{20, 30\}$ and $\mathcal{Z} = \{0, 1\}$. Then

$$\mathcal{X} \times \mathcal{Z} = \{(20, 0), (20, 1), (30, 0), (30, 1)\}$$

.
Assume $\mathcal{X} = \mathbb{R}$ and $\mathcal{Z} = \mathbb{R}$. Then $\mathcal{X} \times \mathcal{Z} = \mathbb{R}^2$.

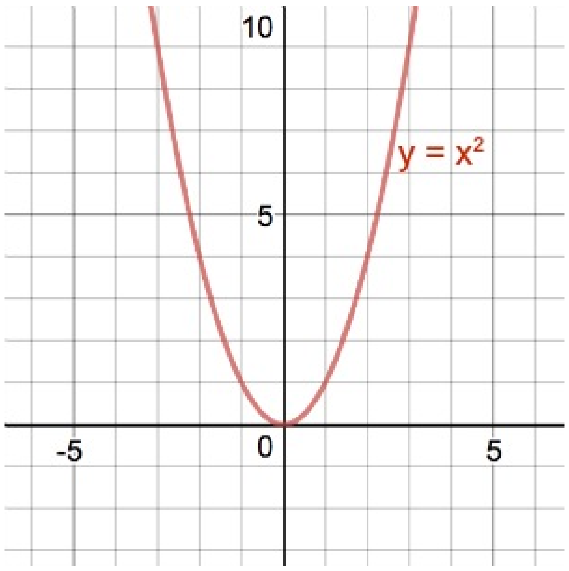
1.2 Functions

- A function f maps a value in the function's *domain* $\mathcal{X}$ to a (unique) value in the function's *co-domain/range* $\mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ can be discrete or continuous sets.
- We write the statement that f is a function from $\mathcal{X}$ to $\mathcal{Y}$ as

$$f : \mathcal{X} \rightarrow \mathcal{Y}$$

Examples:

- $y = x^2$ maps every value in domain $\mathcal{X} \in \mathbb{R}$ to range $\mathcal{Y} \in \mathbb{R}^+$


10
5
-5
0
5
y = x²

2 Probability distributions

A probability distribution is a mathematical function that gives the probability of the occurrence of a set of possible outcomes. The set of possible outcomes is called the *sample space*, which can be discrete or continuous, and is denotes by $\mathcal{X}$. For example, for flipping a coin $\mathcal{X} = \{\text{heads}, \text{tails}\}$. When we actually sample the variable, we get a particular value $x \in \mathcal{X}$. For example, for the first coin flip $x_1 = \text{heads}$. Before we actually sample the outcome, the particular outcome value is still unknown. We say that it is a *random variable*, denoted by X, which always has a associated probability distribution $p(X)$.

Sample space (a set)	$\mathcal{X}$
Random variable	X
Particular value	x

Depending on whether the sample space is a discrete or continuous set, the distribution $p(X)$ and the way to represent it differ. We detail both below.

2.1 Discrete probability distributions

- A *discrete variable* X can take values in a discrete set $\mathcal{X} = \{1, 2, .., n\}$. A particular value that X takes is denoted by x.
- Discrete variable X has an associated *probability mass function*: $p(X)$, where $p : \mathcal{X} \to [0, 1]$.
  - Each possible value x that the variable can take is associated with a probability $p(X = x) \in [0, 1]$.
  - For example, $p(X = 1) = 0.2$, i.e., the probability that X is equal to 1 is 20%.
- Probability distributions always sum to 1, i.e.: $\sum_{x \in \mathcal{X}} p(x) = 1$.

Parameters We represent a probability distribution with *parameters*. For a discrete distribution of size n, we need $n-1$ parameters, i.e., $\{p_{x=1}, .., p_{x=n-1}\}$, where $p_{x=1} = p(x=1)$. The probability of the last category follows from the sum to one constraint, i.e., $p_{x=n} = 1 - \sum_{i=1}^{n-1} p_{x=i}$.

Example: A discrete variable X that can take three values ($\mathcal{X} = \{1, 2, 3\}$), with associated probability distribution $p(X = x)$:

$p(X = 1)$	$p(X = 2)$	$p(X = 3)$
0.2	0.4	0.4

Representing discrete random variables It is important to realize that we always represent a discrete variable *as a vector* of probabilities. Therefore, the above variable X does not really take values $\mathcal{X} = \{1, 2, 3\}$, because 1, 2 and 3 are arbitrary categories (i.e., category two is not twice as much as the first category). We could just as well have written $\mathcal{X} = \{a, b, c\}$. Always think of the possible values of a discrete variable as separate entries. Therefore, we should represent the value of a discrete variable as a vector of probabilities. In the data, when we observe the ground truth, this becomes a *one-hot encoding*, where we put all mass on the observed class.

Example: In the above example, we had ($\mathcal{X} = \{1, 2, 3\}$). Imagine we sample X three times and observe 1, 2 and 3, respectively. We would actually represent these observations as

Observed category	Representation
1	$(1, 0, 0)$
2	$(0, 1, 0)$
3	$(0, 0, 1)$

2.2 Continuous probability distributions

- A *continuous variable* X can take values in a continuous set, e.g., $\mathcal{X} = \mathbb{R}$ (the real line), or $\mathcal{X} = [0, 1]$ (a bounded interval).
- Continuous variable X has an associated *probability density function*: $p(X)$, where $p : \mathcal{X} \to \mathbb{R}^+$ (a positive real number).
- In a continuous set, there are infinitely many values that the random value can take. Therefore, the *absolute* probability of any particular value is 0.
- We can only define absolute probability on an interval, i.e., $p(a < X \leq b) = \int_a^b p(x)$.

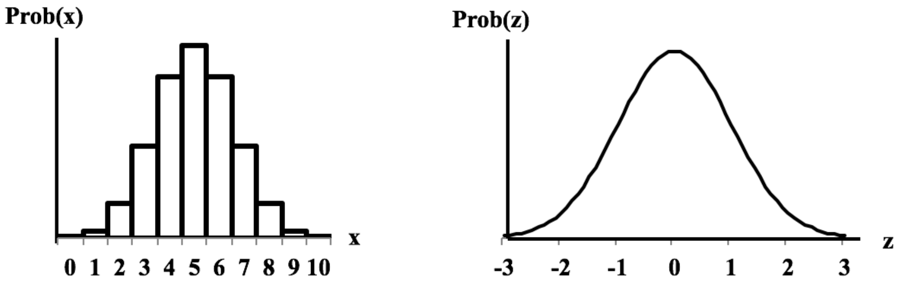


Figure 1: Examples of discrete (left) versus continuous (right) probability distibution.

- For example, $p(2 < X \leq 3) = 0.2$, i.e., the probability that X will fall between 2 and 3 is equal to 20%.

- The interpretation of an individual value of the density, like $p(X = 3) = 4$, is only a *relative* probability. The higher the probability $p(X = x)$, the higher the relative chance that we would observe x.
- Probability distributions always sum to 1, i.e.: $\int x \in \mathcal{X} p(x) = 1$ (not that this time we integrate instead of sum!).

Parameters We need to represent a continuous distribution with a parameterized function, that for every possible value in the sample space predicts a relative probability. Moreover, we need to obey the sum to one constraint. Therefore, there are many *parameterized continuous probability densities*. An example is the Normal distribution. A continuous density is a function $p : \mathcal{X} \to \mathbb{R}^+$ that depends on some parameters. Scaling the parameters allows variation in the location where we put probability mass.

Table 1: Comparison of discrete and continuous probability distributions.

	Discrete distribution	**Continuous distribution**
Input/sample space	Discrete set, e.g. $\mathcal{X} = \{0, 1\}$, with size $n = \|\mathcal{X}\|$	Continuous set, e.g. $\mathcal{X} = \mathbb{R}$
Probability function	Probability mass function (pmf) $p : \mathcal{X} \to [0, 1]$ such that $\sum_{x \in \mathcal{X}} p(x) = 1$	Probability density function (pdf) $p : \mathcal{X} \to \mathbb{R}$ such that $\int_{x \in \mathcal{X}} p(x) = 1$
Possible parametrized distributions	Various, but only need simple Discrete	Various, e.g. Normal, Logistic, Beta, etc.
Parameters	$\{p_{x=1}, .., p_{x=n-1}\}$	Depends on distribution, e.g. for normal: $\{\mu, \sigma\}$
Number of parameters	$n-1 = \|\mathcal{X}\|-1$ (Due to sum to 1 constraint)[a]	Depends on distribution, e.g., for normal: 2
Example distribution function	$p(x = 1) = 0.2$ $p(x = 2) = 0.4$ $p(x = 3) = 0.4$	e.g. for normal $p(x\|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\Big(-\frac{(x-\mu)^2}{2\sigma^2}\Big)$
Absolute probability	$p(x = 1) = 0.2$	$p(3 \leq x < 4) = \int_3^4 p(x) = 0.3$ (on interval)[b]
Relative probability	-	$p(x = 3) = 7.4$

[a]Due to the sum to 1 constraint, we need one parameter less than the size of the sample space, since the last probability is 1 minus all the others: $p_n = 1 - \sum_{i=1}^{n-1} p_i$.

[b]Note that for continuous distributions, probabilities are only defined on *intervals*. The density function $p(x)$ only gives relative probabilities, and therefore we may have $p(x) > 1$, like $p(x = 3) = 5.6$, which is of course not possible (one should not interpret it as an absolute probability). However, $p(a \leq x < b) = \int_a^b p(x) < 1$ by definition.

Example: A variable X that can take values on the real line with distribution

$$p(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\Big(-\frac{(x-\mu)^2}{2\sigma^2}\Big).$$

Here, the mean parameter μ and standard deviation σ are the parameters. We can change them to change the shape of the distribution, while always ensuring that it still sums to one. We draw an example normal distribution in Figure 1, right.
The differences between discrete and continuous probability distributions are summarized in Table 1.

2.3 Conditional distributions

- A conditional distribution means that the distribution of one variable depends on the value that another variable takes.
- We write $p(X|Y)$ to indicate that the value of X depends on the value of Y.

Example: For discrete random variables, we may store a conditional distribution as a table of size $|\mathcal{X}| \times |\mathcal{Y}|$. A variable X that can take three values ($\mathcal{X} = \{1,2,3\}$) and variable Y that can take two values ($\mathcal{Y} = \{1,2\}$). The conditional distribution may for example be:

	$p(X=1\|Y)$	$p(X=2\|Y)$	$p(X=3\|Y)$
$Y=1$	0.2	0.4	0.4
$Y=2$	0.1	0.9	0.0

Note that for each value Y, $p(X|Y)$ should still sum to 1, i.e., it is a valid probability distribution. In the table above, each row therefore sums to 1.

Example: We can similarly store conditional distributions for continuous random variables, this time mapping the input space to the parameters of a *continuous* probability distribution. For example, for $p(Y|X)$ we can assume a Gaussian distribution $\mathcal{N}(\mu(x), \sigma(s))$, where the mean and standard deviation depend on $x \in \mathbb{R}$. Then we can for example specify:

$$\mu(x) = 2x, \quad \sigma(x) = x^2$$

and therefore have

$$p(y|x) = \mathcal{N}(2x, x^2)$$

Note that for each value of X, $p(Y|X)$ still integrates to 1, i.e., it is a valid probability distribution.

2.4 Expectation

We also need the notion of an *expectation.*

2.4.1 Expectation of a random variable

The expectation of a random variable is essentially an average. For a discrete variable, it is defined as:

$$\mathbb{E}_{X\sim p(X)}[f(X)] = \sum_{x\in\mathcal{X}} [x \cdot p(x)] \tag{1}$$

For a continuous variable, the summation becomes integration.

Example:
Assume a given $p(X)$ for a binary variable:

x	$p(X = x)$
0	0.8
1	0.2

The expectation is

$$\mathbb{E}_{X\sim p(X)}[f(X)] = 0.8 \cdot 0 + 0.2 \cdot 1 = 0.2 \tag{2}$$

2.4.2 Expectation of a function of a random variable

More often, and also in the context of reinforcement learning, we will need the expectation *of a function of the random variable*, denoted by $f(X)$. Often, this function maps to a continuous output.

- Assume a function $f : \mathcal{X} \to \mathbb{R}$, which, for every value $x \in \mathcal{X}$ maps to a continuous value $f(x) \in \mathbb{R}$.
- The expectation is then defined as follows:

$$\mathbb{E}_{X\sim p(X)}[f(X)] = \sum_{x \in X} [f(x) \cdot p(x)] \tag{3}$$

For a continuous variable the summation again becomes integration. The formula may look complicated, but it essentially reweights each function outcome by the probability that this output occurs, see the example below.

Example:
Assume a given density $p(X)$ and function $f(x)$:

x	$p(X = x)$	$f(x)$
1	0.2	22.0
2	0.3	13.0
3	0.5	7.4

The expectation of the function can be computed as

$$\begin{aligned}\mathbb{E}_{X\sim p(X)}[f(X)] &= 22.0 \cdot 0.2 + 13.0 \cdot 0.3 + 7.4 \cdot 0.5 \\ &= 12.0\end{aligned}$$

The same principle applies when $p(x)$ is a continuous density, only with the summation replaced by integration.

2.5 Information theory

Information theory studies the amount of information present in distributions, and the way we can compare distributions.

2.5.1 Information

The *information* I of an event x observed from distribution $p(X)$ is defined as:

$$I(x) = -\log p(x).$$

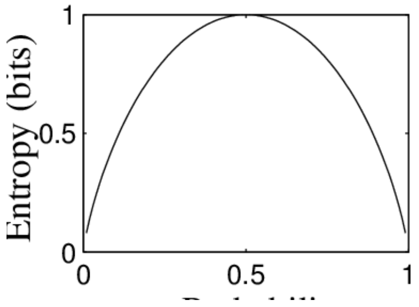


Figure 2: Entropy of a binary discrete variable. Horizontal axis shows the probability that the variable takes value 1, the vertical axis shows the associated entropy of the distribution. High entropy implies high spread in the distribution, while low entropy implies little spread.

In words, the more likeli an observation is (the higher $p(x)$), the less information we get when we actually observe the event. In other words, information of an event is the (potential) reduction of uncertainty. On the two extremes we have:

- $p(x) = 0 \quad : \quad I(x) = -\log 0 = \infty$
- $p(x) = 1 \quad : \quad I(x) = -\log 1 = 0$

2.5.2 Entropy

We define the entropy H of a discrete distribution $p(X)$ as

$$\begin{aligned} H[p] &= \mathbb{E}_{X\sim p(X)}[I(X)] \\ &= \mathbb{E}_{X\sim p(X)}[-\log p(X)] \\ &= -\sum_x p(x)\log p(x) \end{aligned} \tag{4}$$

If the base of the logarithm is 2, then we measure it in *bits*. When the based of the logarithm is e, then we measure the entropy in *nats*. The continuous version of the above equation is called the *continuous entropy* or *differential entropy*.

Informally, the entropy of a distribution is a measure the amount of "uncertainty" in a distribution, i.e., a measure of its "spread". We can nicely illustrate this with a binary variable (0/1), where we plot the probability of a 1 against the entropy of the distribution (Figure 2). We see that on the two extremes, the entropy of the distribution is 0 (no spread at all), while the entropy is maximal for $p(x=1) = 0.5$ (and therefore $p(x=0) = 0.5$), which gives maximal spread to the distribution.

Example: The entropy of the distribution in the previous example is:

$$\begin{aligned} H[p] &= -\sum_x p(x) \log p(x) \\ &= -0.2 \cdot \ln 0.2 - 0.3 \cdot \ln 0.3 - 0.5 \cdot \ln 0.5 = 1.03 \text{ nats} \end{aligned} \tag{5}$$

2.5.3 Cross-entropy

The cross-entropy is defined between two distributions $p(X)$ and $q(X)$ defined over the same support (sample space). The cross-entropy is given by:

$$\begin{aligned} H[p,q] &= \mathbb{E}_{X\sim p(X)}[-\log q(X)] \\ &= -\sum_x p(x) \log q(x) \end{aligned} \tag{6}$$

When we do maximum likelihood estimation in supervised learning, then we actually minimize the cross-entropy between the data distribution and the model distribution (see Appendix IV).

3 Derivative of an expectation

A key problem in gradient-based optimization, which appears all over machine learning, is getting the gradient of an expectation. We will here discuss one well-known method[1]: the *REINFORCE* estimator (in reinforcement learning), which is in other fields also know as the *score function estimator*, *likelihood ratio method*, and *automated variational inference.*

Imagine we are interested in the gradient of an expectation, where the parameters appear in the distribution of the expectation[2]:

$$\nabla_\theta \mathbb{E}_{x\sim p_\theta(x)}[f(x)] \tag{7}$$

We cannot sample the above quantity, because we have to somehow move the gradient inside the expectation (and then we can sample the expectation to evaluate it). To achieve this, we will use a simple rule regarding the gradient of the log of some function $g(x)$:

$$\nabla_x \log g(x) = \frac{\nabla_x g(x)}{g(x)} \tag{8}$$

This results from simple application of the chain-rule.

We will now expand Eq. 7, where we midway apply the above log-derivative trick.

$$\begin{aligned}
\nabla_\theta \mathbb{E}_{x\sim p_\theta(x)}[f(x)] &= \nabla_\theta \sum_x f(x)\cdot p_\theta(x) && \text{definition of expectation}\\
&= \sum_x f(x)\cdot \nabla_\theta p_\theta(x) && \text{push gradient through sum}\\
&= \sum_x f(x)\cdot p_\theta(x)\cdot \frac{\nabla_\theta p_\theta(x)}{p_\theta(x)} && \text{multiply and divide by } p_\theta(x)\\
&= \sum_x f(x)\cdot p_\theta(x)\cdot \nabla_\theta \log p_\theta(x) && \text{log-derivative rule (Eq. 8)}\\
&= \mathbb{E}_{x\sim p_\theta(x)}[f(x)\cdot \nabla_\theta \log p_\theta(x)] && \text{rewrite into expectation}
\end{aligned}$$

[1]Other methods to differentiate through an expectation is through the *reparametrization trick*, as for example used in variational auto-encoders, but we will not further treat this topic here.

[2]If the parameters only appear in the function $f(x)$ and not in $p(x)$, then we can simply push the gradient through the expectation

What the above derivation essential does is *pushing the derivative inside of the sum.* This of course equally applies when we change the sum into an integral. Therefore, for any $p_\theta(x)$, we have:

$$\nabla_\theta \mathbb{E}_{x\sim p_\theta(x)}[f(x)] = \mathbb{E}_{x\sim p_\theta(x)}[f(x)\cdot \nabla_\theta \log p_\theta(x)] \tag{9}$$

This is known as the log-derivative trick, score function estimator, or REINFORCE trick. Although the formula may look complicated, the interpretation is actually really simple. We explain this idea in Figure 3. In the next section we will apply this idea to reinforcement learning.

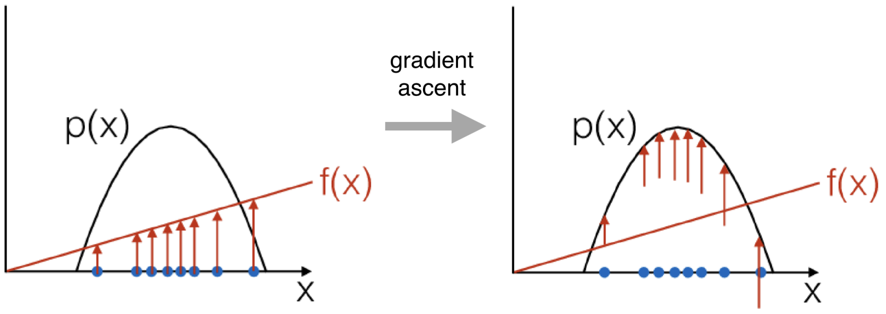


Figure 3: Graphical illustration of REINFORCE estimator. Left: Example distribution $p_\theta(x)$ and function $f(x)$. When we evaluate the expectation of Eq. 9, we take m samples, indicated by the blue dots (in this case $m = 8$). The magnitude of $f(x)$ is shown with the red vertical arrows. Right: When we apply the gradient update, each sample pushes up the density at that location, but the magnitude of the push is multiplied by $f(x)$. Therefore, the higher $f(x)$, the harder we push. Since a density needs to integrate to 1, we will increase the density where we push hardest (in the example on the rightmost sample). The distribution will therefore shift to the right on this update.

Part II
Reinforcement learning problem

We will now formally introduce the Markov Decision Process problem, and discuss how to represent the solution in the form of a policy.

4 Markov Decision Process definition

A Markov Decision Process is a very generic way to define sequential decision-making tasks. We will here focus on continuous MDPs, where the action space (and often the state space) are continuous.

4.1 Definition

The formal definition of a MDP is $\{\mathcal{S}, \mathcal{A}, T(s'|s,a), R(s,a,s'), \gamma, p_0(s_0)\}$, which represent the:

1. **State space**: $\mathcal{S}$.
    - Which observations are possible?
    - *Discrete set*, e.g., $\{0,1\}^{D_S}$, or
    - *Continuous set*, e.g., $\mathbb{R}^{D_S}$ or $[0,1]^{D_S}$.
2. **Action space**: $\mathcal{A}$.
    - Which actions are possible?
    - *Discrete set*, e.g., $\{\text{up}, \text{down}, \text{left}, \text{right}\}$.
    - *Continuous set*, e.g., $[0,1]^{D_A}$.
3. **Transition dynamics**: $T(s'|s,a)$.
    - How does the environment react to an action?
    - *A conditional probability distribution*, represented as a *function* $T : \mathcal{S} \times \mathcal{A} \to p(\mathcal{S})$.
    - In RL usually only available as a simulator.
    - **Terminal states**: some states are terminal. When we reach them, the task ends (i.e., we cannot select a new action, or transition out of them).
4. **Reward function**: $R(s,a,s')$.
    - How rewarding/preferable is each transition?
    - *Function* $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$.
    - Frequently simpler functions, like like $R(s')$ or $R(s,a)$.

- *Cost*, as frequently used in shortest path-problems, is equivalent to *negative reward*. We can formalize all shortest path problems as an MDP, by negating the cost function, or switching from reward maximization to cost minimization.

5. **Discount factor**: γ.
   - How much do we down-weight long-term rewards?
   - A *constant*, $\gamma \in [0, 1]$.
6. **Initial state distribution**: $p_0(s)$.
   - Where do we start?
   - A *distribution* over $\mathcal{S}$.

4.2 Policy

The central question of MDP optimization concerns how we act in the environment. For this we use a *policy* π, which is a *conditional probability distribution* that for each possible state specifies the probability of each possible action. Thereby, it is a mapping from the state space to a probability distribution over the action space:

$$\pi : \mathcal{S} \to p(\mathcal{A})$$

where $p(\mathcal{A}$ can be a discrete or continuous probability distribution. For a particular probability (density) from this distribution we write

$$\pi(a|s)$$

Note that we generally assume a stationary/time-independent policy, i.e., which is the same function for all timepoints (this is valid as long as we use the infinite-horizon cumulative return, which we will encounter later).

Example: For a discrete state space and discrete action space, we may store an explicit policy as a table, e.g.:

s	$\pi(a{=}\text{up}\|s)$	$\pi(a{=}\text{down}\|s)$	$\pi(a{=}\text{left}\|s)$	$\pi(a{=}\text{right}\|s)$
1	0.2	0.8	0.0	0.0
2	0.0	0.0	0.0	1.0
3	0.7	0.0	0.3	0.0
etc.	..	..	..	..

Deterministic policy A special case of a policy is a *deterministic policy*, denoted by

$$\pi(s)$$

where

$$\pi : \mathcal{S} \to \mathcal{A}$$

A deterministic policy selects only a single action in every state. Of course the deterministic action may differ between states, as in the below example:

Example: An example of a deterministic discrete policy is

s	$\pi(a{=}\text{up}\|s)$	$\pi(a{=}\text{down}\|s)$	$\pi(a{=}\text{left}\|s)$	$\pi(a{=}\text{right}\|s)$
1	0.0	1.0	0.0	0.0
2	0.0	0.0	0.0	1.0
3	1.0	0.0	0.0	0.0
etc.	..	..	..	..

We would write $\pi(s = 1) = \text{down}$, $\pi(s = 2) = \text{right}$, etc.

Parametrized policy In practice, we will have to store our policy in memory in some form. This means that our policy will depend on *parameters* θ. To indicate the dependence on policy parameters θ, we will write $\pi_\theta(a|s)$. We extensively discuss policy specification in Chapter 5.

4.3 Traces

We will start interacting with the MDP. At each timestep t we observe s_t, take an action a_t according to policy $\pi(a|s)$, and then observe the next state $s_{t+1} \sim T(\cdot|s, a)$ and reward $r_t = R(s_t, a_t, s_{t+1})$. Repeating this process leads to a *trace* in the environment, which we denote by h_t^n:

$$h_t^n = \{s_t, a_t, r_t, s_{t+1}, .., a_{t+n}, r_{t+n}, s_{t+n+1}\}$$

Here, n denotes the length of the trace. In practice, we often assume $n = \infty$, which means that we run the trace until the domain terminates. In those cases, we will simply write $h_t = h_t^\infty$.

Example: A short trace with three actions could look like:

$$h_0^2 = \{s_0{=}1, a_0{=}\text{up}, r_0{=}\text{-}1, s_1{=}2, a_1{=}\text{up}, r_1{=}\text{-}1, s_2{=}3, a_2{=}\text{left}, r_2{=}20, s_3{=}5\}$$

Distribution over traces Since both the policy and the transition dynamics can be stochastic, we will not always get the same trace from the start state. Instead, we will get a *distribution* over traces. The distribution of traces from the start state (distribution) is denoted by $p_\theta(h_0)$, which depends on the policy parameters θ. The probability of each possible trace from the start is actually given by the product of the probability of each specific transition in the trace:

$$\begin{aligned} p_\theta(h_0) &= p_0(s_0) \cdot \pi(a_0|s_0) \cdot T(s_1|s_0, a_0) \cdot \pi(a_1|s_1)... \\ &= p_0(s_0) \cdot \prod_{t=0}^{\infty} \pi_\theta(a_t|s_t) \cdot T(s_{t+1}|s_t, a_t) \end{aligned} \tag{10}$$

Example: An example trace distribution is visualized in Figure 4.

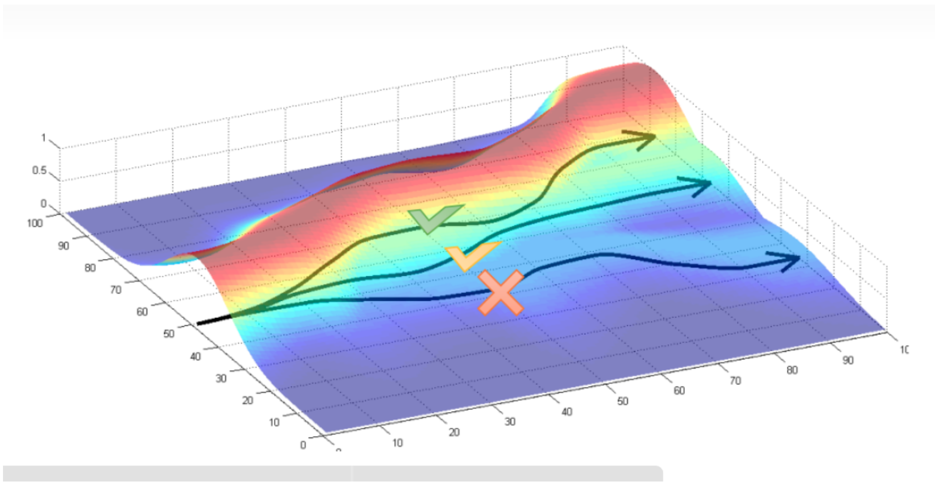


Figure 4: Example distribution over a fictional 2D state space, $\mathcal{S} = [0, 100]^2$, with $p(h)$ shown on the vertical axis. Three example traces sampled from this distribution are shown. The most distant trace had the highest probability under $p(h)$.

4.4 Cumulative reward

We have not yet defined what we actually want to achieve in the sequential decision-making task. The MDP definition included a reward function. We clearly want to achieve as much reward as possible in the task. The sum of all the reward that we achieve is known as the *cumulative reward*, also called the *return*.

We will denote the *return*, or *cumulative reward*, of a trace h_t as:

$$\begin{aligned} R(h_t) &= r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + ... \\ &= \sum_{i=0}^{\infty} \gamma^i r_{t+i} \end{aligned} \tag{11}$$

where $\gamma \in [0, 1]$ denotes a discount factor. It determines how much we down-weight distant rewards. The two extreme cases are:

- $\gamma = 0$: A myopic agent, which only considers the immediate reward, i.e., $R(h_t) = r_t$.
- $\gamma = 1$: A far-sighted agent, which treats all future rewards as equal, i.e., $R(h_t) = r_t + r_{t+1} + r_{t+2} + ...$

Importantly, we cannot use an *infinite-horizon* return (Eq. 11) and $\gamma = 1.0$, since then the cumulative reward may become infinite. In practice, we typically use a discount factor close to 1.0, like $\gamma = 0.99$ or $\gamma = 0.999$.

Example: We use the previous trace example, and assume $\gamma = 0.9$. The cumulative reward is equal to:

$$R(h_0^2) = -1 + 0.9 \cdot -1 + 0.9^2 \cdot 20 = 16.2 - 1.9 = 14.3$$

4.5 Value (expected cumulative reward)

The return of a trace is not the real measure of optimality in which we are interested. The environment can be stochastic, and our policy as well, so for a given policy we do not always observe the same trace. Therefore, we are actually interested in the *average*, or expected, cumulative reward that a certain policy achieves. The average cumulative reward is better known as the *value*. We can define *state values*, and *state-action values*.

State value We define the *state value* $V^\pi(s)$, which is the *average* return we expect to achieve when an agent starts in state s and follows policy π, as:

$$V^\pi(s) = \mathbb{E}_{h_t \sim p(h_t)} \big[\sum_{i=0}^{\infty} \gamma^i \cdot r_{t+i} | s_t = s \big] \tag{12}$$

Example: Imagine we have a policy π, which from state s can result in two traces. The first trace has a cumulative reward of 20, and occurs in 60% of the times. The other trace has a cumulative reward of 10, and occurs 40% of the times. *What is the value of state s?*

$$V^{\pi}(s) = 0.6 \cdot 20 + 0.4 \cdot 10 = 16.$$

Note that 16 is the average return (cumulative reward) that we expect to get from state s under this policy.

Every policy π has only one unique associated value function $V^{\pi}(s)$. We sometimes omit π to simplify notation, simply writing $V(s)$, knowing a state value is always conditioned on a certain policy.

Since the state value is defined for every possible state $s \in \mathcal{S}$, it is really a value *function*, which maps every state to a real number (the average return:

$$V : \mathcal{S} \to \mathbb{R}$$

.

Example: In a discrete state space, the value function can be represented as a table of size $|\mathcal{S}|$.

s	$V^{\pi}(s)$
1	2.0
2	4.0
3	1.0
etc.	etc.

Finally, **the state value of a terminal state is by definition zero**, i.e.,

$$s = \text{terminal} \quad \Rightarrow \quad V(s) := 0.$$

State-action value Instead of state values $V^{\pi}(s)$, we also define state-action values $Q^{\pi}(s, a)$. The only difference is that we now condition on a state *and action*, i.e., we estimate the average return we expect to achieve when taking action a in state s, and then following policy π afterwards:

$$Q^{\pi}(s, a) = \mathbb{E}_{h_t \sim p(h_t)}\big[\sum_{i=0}^{\infty} \gamma^i \cdot r_{t+i} | s_t = s, a_t = a\big] \tag{13}$$

Every policy π has only one unique associated state-action value function $Q^{\pi}(s,a)$. Sometimes we omit π to simplify notation. Again, the state-action value is a *function*

$$Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R},$$

which maps every state-action pair to a real number.

Example: For a discrete state and action space, $Q(s,a)$ can be represented as a table of size $|\mathcal{S}| \times |\mathcal{A}|$. Each table entry stores a $Q(s,a)$ estimate for the specific s,a combination:

	a=up	a=down	a=left	a=right
s=1	4.0	3.0	7.0	1.0
s=2	2.0	-4.0	0.3	1.0
s=3	3.5	0.8	3.6	6.2
etc.	..	..	..	..

The state-action value of a terminal state is by definition zero, i.e.,

$$s = \text{terminal} \quad \Rightarrow \quad Q(s,a) := 0, \quad \forall a$$

A potential benefit of state-action values (Q) versus state values (V) is that state-action values directly tell what every action is worth (which may be useful for action selection).

4.6 Bellman equations

The interesting thing about the value function is that we can write it in *recursive form*, because the value is also defined at the next states.

The Bellman equations for state-values and state-action values are given by:

$$V(s) = \mathbb{E}_{a \sim \pi(\cdot|s)} \mathbb{E}_{s' \sim T(\cdot|a,s)} \Big[r(s,a,s') + \gamma \cdot V(s') \Big]$$

$$Q(s,a) = \mathbb{E}_{s' \sim T(\cdot|a,s)} \Big[r(s,a,s') + \gamma \cdot \mathbb{E}_{a' \sim \pi(\cdot|s')} [Q(s',a')] \Big]$$

Depending on whether the state and action space are discrete or continuous respectively, we may write out these equations differently.

For a discrete state space and discrete action space, we can write out the expectations as summations:

$$V(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \Big[\sum_{s' \in \mathcal{S}} T(s'|s,a) \big[r(s,a,s') + \gamma \cdot V(s') \big] \Big]$$

$$Q(s,a) = \sum_{s' \in \mathcal{S}} T(s'|s,a)\big[r(s,a,s') + \gamma \cdot \sum_{a \in \mathcal{A}} \pi(a|s)[Q(s',a')]\big]$$

For continuous state and action spaces, the summations over policy and transition dynamics are replaced by integration:

$$V(s) = \int_a \pi(a|s)\Big[\int_{s'} T(s'|s,a)\big[r(s,a,s') + \gamma \cdot V(s')\big]\,\mathrm{d}s'\Big]\,\mathrm{d}a$$

The same principle applies to the Bellman equation for state-action values:

$$Q(s,a) = \int_{s'} T(s'|s,a)\big[r(s,a,s') + \gamma \cdot \int_{a'} [\pi(a'|s') \cdot Q(s',a')]\,\mathrm{d}a'\big]\,\mathrm{d}s'$$

We may also have a mixes, e.g., a continuous state space (like visual input) but a discrete action space (like pressing buttons in a game). This would give:

$$Q(s,a) = \int_{s'} T(s'|s,a)\big[r(s,a,s') + \gamma \cdot \sum_{a'} [\pi(a'|s') \cdot Q(s',a')]\big]\,\mathrm{d}s'$$

4.7 Reinforcement learning objective

The objective of reinforcement learning is to achieve the highest possible average return from the start state:

$$J(\pi) = V^{\pi}(s_0) = \mathbb{E}_{h_0 \sim p(h_0|\pi)}\Big[R(h_0)\Big]. \tag{14}$$

for $p(h_0)$ given in Eq. 10. It turns out that there is only one optimal value function, which achieves higher or equal value than all other value functions. We search for a policy that achieves this optimal value function, which we call the optimal policy $\pi^\star$:

$$\pi^\star(a|s) = \arg\max_{\pi} V^{\pi}(s_0) \tag{15}$$

5 Representing the solution

As we have seen in the previous chapter, the eventual goal of reinforcement learning is to find the optimal policy $\pi(a|s)$. We want to store our gradual approximation of this function in memory, which requires a form of supervised learning. We will here focus on *parametric* supervised learning, which uses a parametric function:

$$\pi_\theta : \mathcal{S} \times \Theta \to p(\mathcal{A}).$$

In words, this mapping takes in the state and outputs a probability distribution over the action space. Our aim is to find the best set of parameters θ, i.e., the set of parameters θ that gets π_θ closest to the optimal policy.

In practice, the way we implement π_θ depends on 1) the type of the state space (the input), and 2) the type of action space (the output). We will discuss each separately.

5.1 Dealing with the state space

The type of input space mostly determines what type of supervised learning method we use. We will mostly focus on methods that store a global solution, i.e., a solution for the entire input space. In a parametric approximation, we can either represent the solution as a table, or with function approximation. The choice between tabular or function approximation is actually a topic from supervised learning (See. Part IV). We will summarize the general rules: 1) Tables are exact and easy to implement (especially on discrete input), but they can not generalize, and do not scale to high-dimensions, 2) Function approximation has the benefit of generalization and scale to high-dimensional problems, but are more involved to train and make approximation errors.

We can also use tables in continuous domains, by discretizing the space and treating them as separate bins. Therefore, our choice essentially depends on two characteristics of the state space:

1. The type of space: discrete or continuous.
2. The dimensionality of the space: small or large.

This leads to the following considerations, summarized in Table 3.

- Small, discrete state space: We typically use a table. A good example is tabular Q-learning on a gridworld.

Table 2: Global solution representation approaches depending on state space type (columns) and dimensionality (rows). FA = function approximation.

	Discrete	**Continuous**
Low dimensional	Table(/FA)	Discretization/FA
High dimensional	FA	FA

- High-dimensional state space: For a global solution, we have to use function approximation, due to the curse of dimensionality. A good example is AlphaZero, which used a high-dimensional discrete state space like Chess or Go.
- Small, continuous state space: In a small, continuous state space, we have two options: 1) we can discretize the state space, and use a table, or 2) we can use function approximation. Often we still use function approximation. A good example is CartPole, where we swing up a pole on a cart.
- High-dimensional state space: Function approximation. A good example is robot that learns to pick objects based on visual information.

Local solution Local approaches do not store a policy for the entire state space, but only store a solution for a small subset of the space. This is the common approach in all planning methods. For example, Monte Carlo Tree Search only stores information for the current state and the states that directly descend from it, not for the entire state-space. It searches for a while for a local solution for the current node, and afterwards discards the solution. Local approaches nearly always use tabular solutions.

Note that we can also *combine* the above representations in one algorithm. For example, AlphaZero uses both an MCTS search (a local tabular solution) and a policy/value network (global function approximation).

5.2 Dealing with the action space

The next question is, how do we deal with the type of action space $p(/mathcalA)$ in which we want to predict. There are two important characteristics 1) implicit versus explicit policies, and 2) discrete versus continuous action spaces. We will first discuss implicit policies, see how they are mostly applicable for discrete action spaces, and afterwards discuss explicit policies.

5.2.1 Implicit policy (value-based)

Implicit policies do not directly store $p_\theta(a|s)$, but rather store a value function $Q_\theta(s,a)$ or $V_\theta(s)$. Then, they derive the policy as a direct, hand-coded function from the value function.

$$\pi_\theta(a|s) = f(Q_\theta(s,a))$$

where $f(\cdot)$ is some prespecified function. As we will see below, this mostly works for discrete action spaces.

Implicit discrete policy It turns out that this approach is really useful in *discrete* action spaces. Some examples are:

- The greedy policy, which always selects the action which current has the highest value estimate:

$$\pi_\theta(s) = \arg\max_{a \in \mathcal{A}} Q_\theta(s,a) \tag{16}$$

  The ϵ-greedy policy is a variant where with probability ϵ, we randomly select one of the other actions.

- The Boltzmann policy, which gradually gives higher probability to actions with a higher current value estimate:

$$\pi_\theta(a|s) = \frac{\exp Q_\theta(s,a)/\tau}{\sum_{b \in \mathcal{A}} \exp Q_\theta(s,b)/\tau} \tag{17}$$

  where $\tau \in \mathbb{R}^+$ denotes a temperature parameter, which we can scale to make the policy more or less greedy.

- The UCT policy, well-known from Monte Carlo Tree Search, is also an example of an implicit policy. There are many variants of the UCT formula, but a common form is to use:

$$\pi_\theta(s) = \arg\max_a \left[\bar{Q}_\theta(s,a) + c \cdot \sqrt{\frac{\ln n(s)}{n(s,a)}}\right] \tag{18}$$

  where $\bar{Q}_\theta(s,a)$ is the mean value estimate for the particular state-action pair, $n(s)$ is the number of times we visited node s, and $n(s,a)$ is the number of times we visited action a in node s.

The recurrent theme in implicit policies is that we select actions in the environment based on a hand-designed function from value estimates.

Table 3: Feasible use of implicit and explicit policies (columns) in discrete and continuous action spaces (rows). Implicit policies are really useful in discrete action spaces, but not easy to apply in continuous action spaces.

	Implicit policy $\pi = f(Q_\theta(s,a))$	**Explicit policy** $\pi_\theta(a\|s)$
Discrete action space	✓	✓
Continuous action space	-	✓

Implicit continuous policies While implicit policies are really useful in discrete action spaces, it turns out that they are not very useful in continuous action spaces. Eventually, we want to use our policy to select an action, for example the greedy action. In discrete action spaces, we can quickly evaluate this by simply taking the argmax over the available actions:

$$\pi(s) = \underset{a \in [1,2,..,n]}{\arg\max}\ Q(s,a)$$

which is fast to evaluate. However, in a continuous space, we would for example have to evaluate

$$\underset{a \in \mathbb{R}}{\arg\max}\, Q(s,a)$$

which is a complete optimization problem itself! It would take way too long to find the optimum, while our robot would be waiting. Therefore, for problem with a continuous action space, we actually prefer explicit policies, from which we can always sample.

5.2.2 Explicit policy (policy search)

Explicit policy approaches directly represent a probability distribution over $p(\mathcal{A})$, by predicting the parameters of a distribution over $\mathcal{A}$. Thereby, we can directly sample an action from this distribution when we want to execute the policy.

Explicit discrete policy An explicit discrete policy simply stores the probabilities for every discrete action given a particular state. Two examples were already given in Sec. 4.2. When we use function approximation, we can for example use a neural network that predicts discrete probabilities.

Example: In 19x19 Go we have in principle $19 \cdot 19 = 361$ free positions on the board, i.e., a discrete action space of size 361 (of course, some locations may already be occupied by stones, but we will still predict them, and simply mask them (ignore them) when they are actually proposed). We can the implement a neural network that outputs a vector $\boldsymbol{y} = f_\theta(s)$ of length 361, and use

$$\pi_\theta(a|s) = \text{softmax}(f_\theta(s))$$

for

$$\text{softmax}(y) = \frac{e^{y_i}}{\sum_k e^{y_k}}.$$

The softmax normalizes the vector to probabilities (see Chapter 13 for a recap of supervised learning). Thereby we simply predict the parameters of a discrete probability distribution.

Explicit continuous policy As discussed before, explicit policies are almost unavoidable in problems with a continuous action space. Again, we can simply predict the parameters of an arbitrary continuous probability distribution.

Example: For example, for a one dimensional action space, we may use a neural network to predict a vector of length two, where the first element represents the mean $\mu_\theta(s) = f_\theta(s)$, and use the second element to predict the standard deviation, e.g., $\sigma_\theta(s) = \exp f_\theta(s)$ or $\sigma_\theta(s) = \text{softplus}(f_\theta(s))$, where we exponentiate because the standard deviation needs to be positive. We can then specify an explicit continuous policy as:

$$\pi_\theta(a|s) = \mathcal{N}(a|\mu_\theta(s), \sigma_\theta(s)) = \frac{1}{\sigma_\theta(s)\sqrt{2\pi}} \exp\Big(- \frac{(a-\mu_\theta(s))^2}{2\sigma_\theta^2(s)}\Big).$$

This also generalizes to higher-dimensional continuous action spaces, for example a 2D independent Gaussian for an action space $\mathcal{A} = \mathbb{R}^2$:

$$\pi_\theta(a|s) = \mathcal{N}(\mu_\theta(s), \Sigma_\theta(s))$$

where $\mu = \{\mu_{\theta,1}(s), \mu_{\theta,2}(s)\}$ and $\Sigma = \begin{bmatrix} \sigma_{\theta,1}(s) & 0 \\ 0 & \sigma_{\theta,2}(s) \end{bmatrix}$
Note that the two Gaussians are independent, since the off-diagonal entries in the covariance matrix Σ are 0.

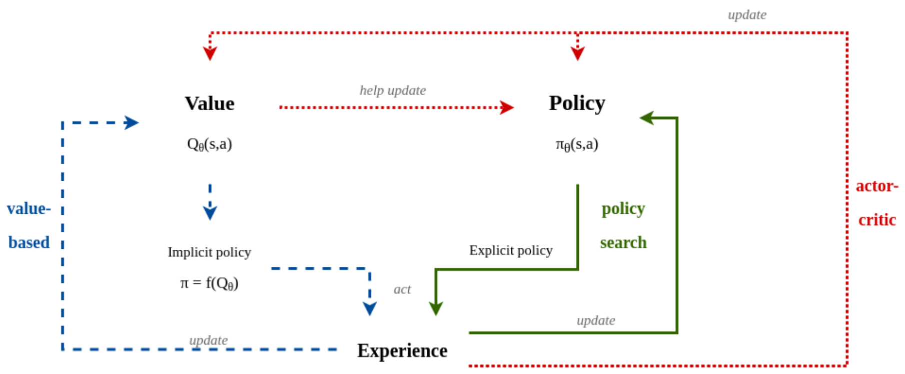


Figure 5: Representing the solution. 1) In *policy search* (green, solid lines), we only represent a policy $\pi_\theta(a|s)$, parametrized by θ. 2) In *value-based methods* (blue, half-dashed), we only represent a value function $Q_\theta(s,a)$. The policy is an implicit function of the value, i.e. $\pi(a|s) = f(Q_\theta(s,a))$. 3) In *actor-critic* methods (red, small-dashed), we use both a policy and value representation, where the value aids in the update of the policy.

For explicit continuous policies, we can represent a deterministic policy in a special form, since we can directly predict a value in the continuous space. You could also think of this as predicting only the mean of a Gaussian distribution.

Example: We implement a deterministic, explicit, continuous policy as

$$\pi_\theta(s) = f_\theta(s),$$

as long as f_θ has the restriction to respect the range of the action space (i.e., if our environment only allows continuous actions between 0 and 1, then our prediction should also output a number between 0 and 1).

5.2.3 Actor-critic

So far, we have looked at value-based methods, which optimize a learned value function (which implicitly defines the policy), and policy-based methods, which directly optimize a learned policy function (although we have not actually discussed how to optimize them). There is a third category of approaches that actually uses *both* a policy and value function, better known as *actor-critic* methods (where the 'actor' is the policy and the 'critic' is the value function). We will later explain in greater detail how a value function may help to update a policy function.

Summary of policy search, value-based RL and actor-critic We shortly summarize these two ways of representing a policy in Figure 5. It shows how we may either store an explicit policy (green, solid lines), an implicit policy in the form of a value function (blue, half-dashed lines), or a combination of both (known as actor-critic, red, small-dashed lines).

Table 4 summarizes some example equations of how different methods represent their solution. Table **??** provides some illustrative examples for each combination.

Table 4: Example solution representations, split up for 1) value-based RL versus policy-based RL (or actor-critic = both), 2) stochastic or deterministic policy, and 3) discrete or continuous action space. In general, we assume $f_/theta(s)$ output a real value, and $f_/theta(s,a)$ a vector of the number of actions. ∘: Value-based approaches are not preferable in continuous action spaces, see text. △: We can specify $\pi_\theta(s) = \arg\max_a f_\theta(s,a)$, but argmax is non-differentiable, and it would only be feasible with gradient-free policy search.

		Discrete action space	**Continuous action space**
Value-based	**Stochastic**	e.g., $\pi_\theta(a\|s) = \epsilon\text{-greedy}(Q_\theta(s,a))$	(-)$^\circ$
	Deterministic	e.g., $\pi_\theta(s) = \arg\max_a Q_\theta(s,a)$	(-)$^\circ$
Policy search	**Stochastic**	e.g. $\pi_\theta(a\|s) = \text{softmax}(f_\theta(s,a))$	e.g., $\pi_\theta(a\|s) = \mathcal{N}(\mu_\theta(s), \sigma_\theta(s))$
	Deterministic	(-)$^\triangle$	e.g., $\pi_\theta(s) = f_\theta(s)$
Actor-critic	**Stochastic**	e.g. $\pi_\theta(a\|s) = \text{softmax}(f_\theta(s,a))$ & $V_\theta(s) = f_\theta(s)$	e.g., $\pi_\theta(a\|s) = \mathcal{N}(\mu_\theta(s), \sigma_\theta(s))$ & $V_\theta(s) = f_\theta(s)$
	Deterministic	(-)$^\triangle$	e.g., $\pi_\theta(s) == f_\theta(s)$ & $V_\theta(s) = g_\theta(s)$

6 Cumulative reward estimation

All RL methods will need an estimate of the cumulative reward (return) from a certain state(-action). Given a trace

$$h_t = \{s_t, a_t, r_t, s_{t+1}, a_{t+1}, r_{t+1}, ..\},$$

all RL approaches estimate the cumulative reward with a variant of the below equation:

$$\hat{Q}_{\text{n-step}}(s_t, a_t) = \sum_{k=0}^{n-1} r_{t+k} + V_\theta(s_{t+n}), \tag{19}$$

where different values of n imply:

- $n = 1$ is a one-step target, where we sample one transition an plug in the learned value function.
- $1 < n < \infty$ is an n-step target, where we take an intermediate number of steps an then bootstrap.
- $n \to \infty$ this is a *Monte Carlo* target, where we never bootstrap.

and $V_\theta(s_{t+n})$ can be

- An **off-policy** target from Q: when $V_\theta(s) = \max_a Q_\theta(s, a)$ **and** $n = 1$.
- An **on-policy** target: when $V_\theta(s) = Q_\theta(s, a)$ for $a \sim \pi_\theta(a|s)$ **and/or** $n > 1$.

These ideas are summarized in Tables 5 and 6. Both the depth n and the use of on-policy or off-policy is subject to a trade-off:

- The bootstrap depth n trades-off between bias and variance. For $n \to \infty$, we have an unbiased, but high-variance estimate (every trace can have a different return, and we only sample a few of the many possible traces). For $n \to 1$, we reduce the variance, since we aggregate different potential traces in the same bootstrapped value estimate. However, we decrease variance but increase bias, since the value estimate can be incorrect.
- On-policy targets follow the behavioural policy and are typically more stable, but when we retain exploration they will never learn the optimal policy or value function. Off-policy targets can learn the optimal policy/value, but they can be unstable due to the max operation, especially in combination with function approximation.

Table 5: On-policy and off-policy following Eq. 19. Off-policy estimates require $n = 1$ **and** a different policy than the behavioural for the bootstrap estimate. When we max, we call the algorithm Q-learning. An on-policy algorithm instead bootstraps the behavioural policy. When we do this after 1-step, we call it SARSA. Any target deeper than depth 1 is by definition on-policy, since the initial steps in the trace are by definition on-policy.

	$V_\theta(s) = \arg\max_a Q_\theta(s, a)$	$V_\theta(s) = Q_\theta(s, a)$
$n = 1$	off-policy	on-policy
$n > 1$	on-policy	on-policy

Table 6: Equations for cumulative reward estimation, splits up for on/off policy, and the roll-out depth. Of course, we can also use $V_\theta(s) = \max_a Q_\theta(s,a)$ and $n > 1$, but it will stay on-policy (so we entered a dash instead).

	On-policy	**Off-policy**
$n = 1$	$\hat{Q}_{\text{SARSA}}(s_t, a_t) = r_t + Q_\theta(s, a_{t+1})$	$\hat{Q}_{\text{Q-learn}}(s_t, a_t) = r_t + \max_{a_{t+1}} Q_\theta(s, a_{t+1})$
$1 < n < \infty$	$\hat{Q}_{\text{n-step}}(s_t, a_t) = \sum_{k=0}^{n-1} r_{t+k} + V_\theta(s_{t+n})$	-
$n = \infty$	$\hat{Q}_{\text{Monte Carlo}}(s_t, a_t) = \sum_{k=0}^{\infty} r_{t+k}$	-

7 Value-based reinforcement learning

We give a very brief summary of value-based reinforcement learning, where we want to update a value function (implicit policy) from data. As mentioned before, this is mostly applicable to discrete action space problems. In value-based RL, we often prefer to learn a state-action value function $Q_\theta(s, a)$, since we can directly use it to act. In actor-critic approaches (see Sec. 10), we often learn a state value function $V_\theta(s)$ instead, for bootstrapping or baseline subtraction. We will here only discuss the value-based RL approach that uses a state-action value $(Q_\theta(s, a))$.

7.1 Value function update

Given a cumulative reward estimate, we can update the learned value function, for example based on a squared loss:

$$L(\theta|s_t, a_t) = \mathbb{E}_{h_0 \sim p_\theta(h_0)} \left[\left(Q_\theta(s_t, a_t) - \hat{Q}_{\text{n-step}}(s_t, a_t) \right)^2 \right]$$

with $\hat{Q}_{\text{n-step}}(s_t, a_t)$ as described in Eq. 19.

7.2 Exploration

We need to ensure exploration while collecting data in the environment. For this, we specify a behavioural policy:

$$\pi_\theta(a|s) = g(Q_\theta(s, a))$$

for some function $g()$. Simple approaches to ensure exploration inject noise into the behavioural policy, i.e., not always selecting the action which currently has the highest value estimate. Some examples of exploratory policies for value-based RL in discrete action spaces are:

- The ϵ-greedy policy

$$\pi_\theta(a|s) = \begin{cases} 1.0 - \epsilon, & \text{if } a = \arg\max_{b \in \mathcal{A}} Q_\theta(s, b) \\ \epsilon/(|\mathcal{A}| - 1), & \text{otherwise} \end{cases} \tag{20}$$

  In words, we select with small probability ϵ a random action, which ensures exploration.

- The Boltzmann policy (already encountered)

$$\pi_\theta(a|s) = \frac{\exp Q_\theta(s,a)/\tau}{\sum_{b\in\mathcal{A}} \exp Q_\theta(s,b)/\tau} \tag{21}$$

  where $\tau \in \mathbb{R}^+$ denotes a temperature parameter, which we can scale to make the policy more or less greedy. This approach gradually gives higher probability to actions with a higher current value estimate, but still ensures exploration of other actions than the greedy one.

An example algorithm for Q-learning is shown in Alg. 1.

Algorithm 1: Q-learning with function approximation

```
Input: A parametrized state-action value function $Q_\theta(s,a)$, number of
       episodes $n$, learning rate $\eta \in \mathbb{R}^+$.
Initialization: Randomly initialize parameters $\theta$.
for $i = 1..n$ do
    Sample initial state $s_0 \sim p_0(s_0)$
    grad = $\mathbf{0}$
    for $t = 0..K$ do
        Sample action $a_t \sim f(Q_\theta(s_t, \cdot))$              /* Sample a trace */
        Observe from environment $(r_t, s_{t+1})$
        $\hat{Q}(a_t, s_t) = r_t + \max_{a_{t+1}} Q_\theta(s_{t+1}, a_{t+1})$  /* Off-policy estim */
        grad += $\frac{d}{d\theta}\left(Q_\theta(s_t,a_t) - \hat{Q}_{\text{n-step}}(s_t,a_t)\right)^2$    /* Squared loss */
    end
    $\theta \leftarrow \theta - \eta \cdot$ grad                          /* Update params */
end
$\pi_{\text{greedy}}(s,a) = \arg\max_a Q(s,a) \quad \forall s,a$              /* Greedy policy */
Return $Q_\theta(s,a)$ or $\pi_\theta(s,a)$.
```

Part III

Policy-based RL

We will now go into detail on methods that directly optimize an explicit policy.

8 Policy search

Approaches that directly optimize an explicit policy are called *policy search.* We can interpret the reinforcement learning objective as a direct optimization problem in policy space, since we want to find

$$\theta^\star = \arg\max_\theta J(\theta). \tag{22}$$

In our case, $J(\theta)$ is specified in Eq. 14. There are two main approaches to such optimization problems:

- **Gradient-based optimization** (= policy gradients): These methods use the derivative of the objective to find the optimum. In the case of a maximization, we may then apply *gradient ascent*, which in each iteration i of the algorithm makes the following update:

  $$\theta_{i+1} = \theta_i + \eta \cdot \nabla_\theta J(\theta)$$

  for learning rate $\eta \in \mathbb{R}^+$. A full algorithm is given in Alg. 2.

Algorithm 2: Gradient ascent optimization

```
Input: A differentiable objective $J(\theta)$, learning rate $\eta \in \mathbb{R}^+$, threshold
       $\epsilon \in \mathbb{R}^+$.
Initialization: Randomly initialize $\theta$ in $\mathbb{R}^d$.
repeat
  |  $\theta \leftarrow \theta + \eta \cdot \nabla_\theta J(\theta)$
until $\theta$ converges;
Return Optimal parameters $\theta$
```

- **Gradient-free optimization**: Examples of gradient-free optimization strategies include evolutionary strategies (ES), like the *cross-entropy method* (CEM). These can actually be very efficient and are relatively easy to implement, so they can serve as a useful baseline. We will shortly discuss these after the gradient-based policy search methods, in Chapter 11.

We will focus on gradient-based approaches to policy search, which are known as *policy gradients.* However, to derive the policy gradient, we first need to understand the derivative of an expectation.

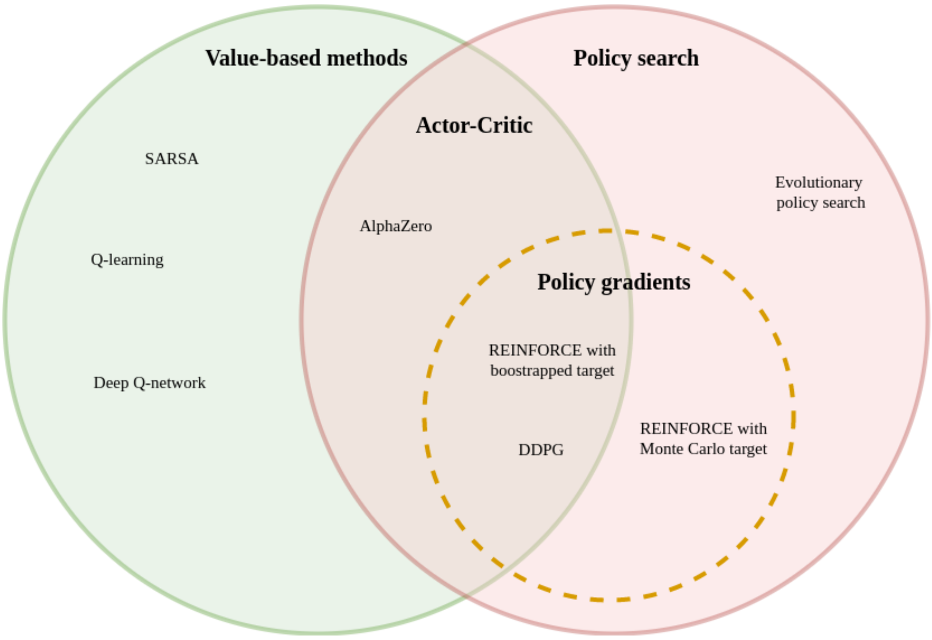


Figure 6: Graphical illustration of value-based RL, policy-based RL and actor-critic (value + policy). We also depict policy gradient methods, which use gradient-based optimization for the policy update, which can be combined with a value function or not.

9 Gradient-based policy search: policy gradients

We will now derive the policy gradient equation. The question is: can we derive the gradient of the reinforcement learning objective (Eq. 14). In this case, we are looking for the following derivative:

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{h_0 \sim p_\theta(h_0)} \Big[R(h_0) \Big] \tag{23}$$

The problem is that this is not a quantity that we can sample. We would like to get the gradient inside the expectation, so that we can sample data, and then compute the gradient.

9.1 Monte Carlo Policy Gradient (REINFORCE)

We will now show the main derivation of the Monte Carlo Policy Gradient, better known as the REINFORCE estimator. Carefully read Sec. 3, which introduces the the log-derivative trick to differentiate through an expectation. With basic calculus it shows the following identity (Eq. 9):

$$\nabla_\theta \mathbb{E}_{x \sim p_\theta(x)}[f(x)] = \mathbb{E}_{x \sim p_\theta(x)}[f(x) \cdot \nabla_\theta \log p_\theta(x)] \tag{24}$$

We can directly apply this idea to $\nabla_\theta J(\theta)$:

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta \mathbb{E}_{h_0 \sim p_\theta(h_0)} \Big[R(h_0) \Big] \\ &= \mathbb{E}_{h_0 \sim p_\theta(h_0)} \Big[R(h_0) \cdot \nabla_\theta \log p_\theta(h_0) \Big] \quad \text{Log-derivative trick (Eq. 9)}\end{aligned} \tag{25}$$

Now, we still need to simplify the log-derivative of a trace distribution (defined in Eq. 10):

$$
\begin{aligned}
\nabla_\theta \log p_\theta(h_0) &= \nabla_\theta \log \Big[p_0(s_0) \prod_{t=0}^{n} \pi_\theta(a_t|s_t) \cdot T(s_{t+1}|s_t, a_t) \Big] && \text{Definition of } p_\theta(h_0) \\
&= \nabla_\theta \Big[\log p_0(s_0) + \sum_{t=0}^{n} \log \pi_\theta(a_t|s_t) + \sum_{t=0}^{n} \log T(s_{t+1}|s_t, a_t) \Big] && \text{Log of product} \\
&= \sum_{t=0}^{n} \nabla_\theta \log \pi_\theta(a_t|s_t) && \text{Dependence on } \theta
\end{aligned}
\tag{26}
$$

Putting the above expression in, we get:

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \mathbb{E}_{h_0 \sim p_\theta(h_0)} \Big[R(h_0) \Big] \\
\nabla_\theta J(\theta) &= \mathbb{E}_{h_0 \sim p_\theta(h_0)} \Big[R(h_0) \cdot \nabla_\theta \log p_\theta(h_0) \Big] && \text{Log-derivative trick (Eq. 9)} \\
\nabla_\theta J(\theta) &= \mathbb{E}_{h_0 \sim p_\theta(h_0)} \Big[R(h_0) \cdot \sum_{t=0}^{n} \nabla_\theta \log \pi_\theta(a_t|s_t) \Big] && \text{Log derivative of trace (Eq. 26)}
\end{aligned}
\tag{27}
$$

The above formulation essentially says that we should sample traces h, and for step in the trace multiply the gradient of the policy with the return of the whole trace, $R(h_0)$. In practice, the gradient of an action at a particular timestep only depends on what happens *after* that decision. Therefore, a more common formulation of the policy gradient only considers the remaining return from the specific timepoint, i.e.:

$$
\nabla_\theta \mathbb{E}_{h_0 \sim p_\theta(h_0)} \Big[R(h_0) \Big] = \mathbb{E}_{h_0 \sim p_\theta(h_0)} \Big[\sum_{t=0}^{n} R(h_t) \nabla_\theta \log \pi_\theta(a_t|s_t) \Big] \tag{28}
$$

This equation is known as the *policy gradient theorem*. In particular, we call the above version the *Monte Carlo policy gradient*, or *REINFORCE* estimator.

It is called a Monte Carlo estimator because we sample entire traces in the environment. In practice, we would estimate the above quantity by sampling M traces in the environment, i.e. $\{h_0^i\}_{i=1}^M$ and computing:

$$
\nabla_\theta J(\theta) \approx \frac{1}{M} \sum_{i=1}^{M} \Big[\sum_{t=0}^{n} R(h_t^i) \nabla_\theta \log \pi_\theta(a_t|s_t) \Big] \tag{29}
$$

Note that, when we use automatic differentiation software (like Tensorflow or PyTorch), we would implement the MC policy gradient by minimizing the loss

$$L(\theta) = -\frac{1}{M}\sum_{i=1}^{M}\Big[\sum_{t=0}^{n} R(h_t^i)\log \pi_\theta(a_t|s_t)\Big] \tag{30}$$

where we put a minus in front since it is now a loss (we minimize something), and automatically differentiating the above expression gives the correct gradient estimate.

The implementation of the Monte Carlo policy gradient is shown in Algorithm 3.

Algorithm 3: Monte Carlo policy gradient (REINFORCE)

```
Input: A differentiable policy $\pi_\theta(a|s)$, parametrized by $\theta \in \mathbb{R}^d$.
A learning rate $\eta$.
Initialization: Randomly initialize $\theta$ in $\mathbb{R}^d$.
while not converged do
    grad ← 0
    for $m \in 1,..,M$ do
        Sample trace $h_0 = \{s_0, a_0, r_0, s_1, .., s_{n+1}\}$ following $\pi_\theta(a|s)$
        $R \leftarrow 0$
        for $t \in n,..,1,0$ do
            $R \leftarrow r_t + \gamma \cdot R$                 /* Backwards through trace */
            grad += $R \cdot \nabla_\theta \log \pi_\theta(a_t|s_t)$  /* Add to total gradient */
        end
    end
    $\theta \leftarrow \theta + \eta \cdot$ grad
end
Return $\pi_\theta(a|s)$
```

Practical interpretation of policy gradient Sometimes a complicated equation and derivation can have a very simple interpretation. We already saw the interpretation of the reinforce estimator in Figure 3. Essentially, the reinforce estimator says that we should push up the probability of each action, but multiply the strength of the push by the return for that action.

Example: We have a policy network $\pi_\theta(a|s) = \text{softmax}(f_\theta(s))$ that maps into a discrete action space. At a particular state s, we have three available actions (a_1,, a_2 and a_3), with current probabilities of 0.2, 0.5 and 0.3, respectively. We manage to sample a trace from each available action, which gives returns of 65, 70 and 75, respectively. The returns indicate that we should multiply the gradient of θ_{a_3} by the largest value, i.e., this action should increase the most. For example, after an update with a large learning rate we could get probabilities of 0.2, 0.35 and 0.45.

9.2 Exploration

Just as with value-based reinforcement learning, we still need to ensure exploration pressure, i.e., give the incentive to sometimes try an action which currently seems suboptimal.

Deterministic policy & noise When we learn a deterministic policy $\pi_\theta(s)$, then we can manually add exploration noise. In a continuous action space we could use Gaussian noise, while in a discrete action space we can use Dirichilet noise. For example, in a 1D continuous action space we could use:

$$\pi_{\theta,\text{behaviour}}(a|s) = \pi_\theta(s) + \mathcal{N}(0, \sigma),$$

where σ is an exploration hyperparameter.

Stochastic policy & entropy regularization When we learn a stochastic policy $\pi(a|s)$, then exploration is already partially ensured due to the stochastic nature of our policy. For example, when we predict a Gaussian distribution, then simply sampling from this distribution will already induce variation in the chosen actions.

$$\pi_{\theta,\text{behaviour}}(a|s) = \pi_\theta(a|s)$$

However, one potential problem is *collapse* of the policy distribution. The distribution then becomes too narrow, and we lose exploration pressure.

Although we could simply add additional noise (as mentioned above), another common approach is to use *entropy regularization* (See. Chapter 2 for details on entropy). We then add an additional penalty to the loss function, that enforces the entropy of the distribution to stay larger, i.e., that enforces the distribution to stay wide.

For example, we could extend the policy gradient equation (Eq. 36) to

$$\nabla_\theta J(\theta) = \mathbb{E}_{h_0 \sim p_\theta(h_0)} \Big[\sum_{t=0}^{n} R_t \nabla_\theta \log \pi_\theta(a_t|s_t) + \eta \nabla_\theta H[\pi_\theta(a|s)] \Big] \tag{31}$$

where $\eta \in \mathbb{R}^+$ is a constant that scales the amount of entropy regularization. This ensures that we will move $\pi_\theta(a|s)$ towards the optimal policy, while also ensuring that the policy stays as wide as possible (essentially trading both off against eachother).

10 Actor-critic

We will discuss two types of actor-critic approaches. First, we will discuss approaches that stabilize the policy gradient theorem update through the use of a value function. Next, we will also shortly discuss an alternative way to get a policy gradient, known as deterministic policy gradient.

10.1 Actor-critic policy gradient

The MC policy gradient is unbiased, as we previously established from our derivation. However, it has high variance (the size and direction of the update can strongly vary over different samples), which can originate from two sources: 1) high variance in the cumulative reward estimate, and 2) high variance in the gradient estimate. The solution for each problem are a) bootstrapping, and b) baseline subtraction. Both of these methods use a learned value function, which we denote by $V_\phi(s)$.

10.1.1 Bootstrapping

The policy gradient is exact when we sample all possible traces. In practice, there are exponentially many traces possible for a given stochastic policy, and we cannot afford to all sample them for one update. Therefore, in practice we usually keep M small, and sometimes even use $M = 1$, i.e., update from a single trace. In those cases, the update that we compute may strongly differ between traces, and in many of the traces we do not even hit a high reward region. In other words, the update has high variance. A practical solution to this problem is bootstrapping, which trades-off bias for variance. We already explained this topic in Sec. 6.

In short, we can use bootstrapping to compute an n step target:

$$\hat{Q}_{\text{n}}(s_t, a_t) = \sum_{k=0}^{n-1} r_{t+k} + V_\phi(s_{t+n}) \tag{32}$$

we can then update the value function, for example on a squared loss

$$L(\phi|s_t, a_t) = \big(\hat{Q}_n(s_t, a_t) - V_\phi(s_t)\big)^2 \tag{33}$$

and update the policy with the standard policy gradient

$$\nabla_\theta L(\theta|s_t, a_t) = \hat{Q}_n(s_t, a_t) \cdot \nabla_\theta \log \pi_\theta(a_t|s_t)] \tag{34}$$

An example algorithm is shown in Alg. 4.

Algorithm 4: Actor-critic policy gradient with bootstrapping. In practice, one would often average the gradient over multiple episodes to increase stability.

```
Input: A policy $\pi_\theta(a|s)$, a value function $V_\phi(s)$
A estimation depth $n$, learning rate $\eta$.
Initialization: Randomly initialize $\theta$ and $\phi$.
while not converged do
    Sample trace $h_0 = \{s_0, a_0, r_0, s_1, .., s_{T+1}\}$ following $\pi_\theta(a|s)$
    for $t = 0..T$ do
        $\hat{Q}_n(s_t, a_t) = \sum_{k=0}^{n-1} r_{t+k} + V_\theta(s_{t+n})$          /* $n$-step target */
    end
    $\phi \leftarrow \phi - \eta \cdot \sum_t \nabla_\phi(\hat{Q}_n(s_t, a_t) - V_\phi(s_t))^2$  /* Descent value loss */
    $\theta \leftarrow \theta + \eta \cdot \sum_t [\hat{Q}_n(s_t, a_t) \cdot \nabla_\theta \log \pi_\theta(a_t|s_t)]$  /* Ascent pol grad */
end
Return $\pi_\theta(a|s)$
```

10.1.2 Baseline subtraction

The policy gradient can also be numerically unstable. Imagine, in a given state with three available actions, we sample actions returns of 65, 70, and 75, respectively. We then implement the policy gradient equation, which will try to push the probability of each action up, since the return for each action is positive. The above method may lead to a problem, since we are pushing all actions up (just harder on one of them). It might be better when we push up on actions that are better than average, and push down on average that are below average. We can do so through *baseline substraction.*

The most common choice for the baseline is the value function. When we subtract the value from a state-action value estimate, we get the *advantage function*:

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t).$$

Compared to the cumulative return, the advantage function subtracts the value of the state s. Thereby, it estimates how much better a particular action is compared to what we expect to get on average from a particular state. In practice, we can use any of the previous methods to estimate the cumulative reward $\hat{Q}(s_t, a_t)$, compute

$$\hat{A}_{\mathrm{n}}(s_t, a_t) = \hat{Q}_n(s_t, a_t) - V_\phi(s_t).$$

and update the policy with

Algorithm 5: Actor-critic policy gradient with bootstrapping and baseline subtraction. In practice, one would often average the gradient over multiple episodes to increase stability.

```
Input: A policy $\pi_\theta(a|s)$, a value function $V_\phi(s)$
A estimation depth $n$, learning rate $\eta$.
Initialization: Randomly initialize $\theta$ and $\phi$.
while not converged do
    Sample trace $h_0 = \{s_0, a_0, r_0, s_1, .., s_{T+1}\}$ following $\pi_\theta(a|s)$
    for $t = 0..T$ do
        $\hat{Q}_n(s_t, a_t) = \sum_{k=0}^{n-1} r_{t+k} + V_\theta(s_{t+n})$          /* $n$-step target */
        $\hat{A}_\mathrm{n}(s_t, a_t) = \hat{Q}_n(s_t, a_t) - V_\phi(s_t)$              /* Advantage */
    end
    $\phi \leftarrow \phi - \eta \cdot \sum_t \nabla_\phi (\hat{Q}_n(s_t, a_t) - V_\phi(s_t))^2$  /* Descent value loss */
    $\theta \leftarrow \theta + \eta \cdot \sum_t [\hat{A}_n(s_t, a_t) \cdot \nabla_\theta \log \pi_\theta(a_t|s_t)]$  /* Ascent pol grad */
end
Return $\pi_\theta(a|s)$
```

$$\nabla_\theta L(\theta|s_t, a_t) = \hat{A}_n(s_t, a_t) \cdot \nabla_\theta \log \pi_\theta(a_t|s_t)] \tag{35}$$

A full algorithm is shown in Alg. 5.

10.1.3 Generic policy gradient formulation

With the two above ideas we can actually formulate an entire spectrum of policy gradient methods, depending on the type of cumulative reward estimate they use. In general, the policy gradient estimator takes the following form, where we now introduced a new target Φ_t:

$$\nabla_\theta J(\theta) = \mathbb{E}_{h_0 \sim p_\theta(h_0)} \Big[\sum_{t=0}^{n} \Psi_t \nabla_\theta \log \pi_\theta(a_t|s_t) \Big] \tag{36}$$

There are a variety of potential choices for Ψ_t, based on the potential use of bootstrapping and baseline substraction:

$$\begin{aligned}
\Psi_t &= \hat{Q}_{MC}(s_t, a_t) = \sum_{i=t}^{\infty} \gamma^i \cdot r_i && \text{Monte Carlo target} \\
\Psi_t &= \hat{Q}_n(s_t, a_t) \quad = \sum_{i=t}^{n-1} \gamma^i \cdot r_i + \gamma^n V_\theta(s_n) && \text{bootstrap (n-step target)} \\
\Psi_t &= \hat{A}_{MC}(s_t, a_t) = \sum_{i=t}^{\infty} \gamma^i \cdot r_i - V_\theta(s_t) && \text{Baseline subtraction} \\
\Psi_t &= \hat{A}_n(s_t, a_t) \quad = \sum_{i=t}^{n-1} \gamma^i \cdot r_i + \gamma^n V_\theta(s_n) - V_\theta(s_t) && \text{Baseline + bootstrap} \\
\Psi_t &= Q_\theta(s_t, a_t) && \text{Q-value approximation}
\end{aligned}$$

10.2 Deterministic Policy Gradient

As a second approach, we may also use a learned value function as a differentiable target to optimize the policy against, i.e., let the policy follow the value function. An example is the *deterministic policy gradient*. We will introduce ϕ for the parameters of the value function, to discriminate them from the policy parameters. Imagine we collect data $\mathcal{D}$ and train a value network $Q_\phi(s, a)$ following Chapter 7. We can then attempt to optimize the parameters of a deterministic policy by simply optimizing the prediction of the value network:

$$J(\theta) = \mathbb{E}_{s \sim \mathcal{D}} \Big[\sum_{t=0}^{n} Q_\phi(s, \pi_\theta(s)) \Big],$$

which by the chain-rule gives the following gradient expression

$$\nabla_\theta J(\theta) = \mathbb{E}_{s \sim \mathcal{D}} \Big[\sum_{t=0}^{n} \nabla_a Q_\phi(s, a) \cdot \nabla_\theta \pi_\theta(s) \Big]. \tag{37}$$

In essence, we first train a state-action value network based on sampled data, and then *let the policy follow the value network*, by simply chaining the gradients. Thereby, we push the policy network in the direction of those actions a that increase the value network prediction, i.e., towards actions that perform better.

11 Gradient-free policy search

A very simple approach to gradient free policy optimization is shown in Algorithm 6, better known as the cross-entropy method. The idea is straighforward: we 1) initialize a Gaussian distribution over the parameter vector, 2) sample a set of parameter realizations from this distribution, and evaluate their cumulative reward, 3) we select the elite u set, i.e., the set of policy parameters which achieved the best $u\%$ performance, 4) we refit the Gaussian distribution to the elite set, and 5) repeat this procedure for a predetermined number of iterations. This is an example of an evolutionary algorithm, where we repeatedly adjust the population to the samples which performed best in the previous iteration.

Algorithm 6: Cross-entropy method (CEM) for reinforcement learning

```
Input: A differentiable policy $\pi_\theta(a|s)$, parametrized by $\theta \in \mathbb{R}^d$, where
       $\theta \sim \mathcal{N}(\mu, \sigma)$. Number of iterations $n_{\text{iter}}$, number of samples
       $n_{\text{sample}}$, top percentage $u$.
Initialization: Randomly initialize $\mu_1 \in \mathbb{R}^d$ and $\sigma_1 \in (\mathbb{R}^+)d$.
for $i = 1..n_{iter}$ do
    Sample parameters $\theta_j \sim \mathcal{N}(\mu_i, \text{diag}(\sigma_i))$ for $j = 1..n_{\text{sample}}$
    Sample returns $R_j \sim \pi_{\theta_j}(a|s)$
    Select elite set of $\theta_j$ giving the top $u\%$ returns $R_j$
    $\mu_{i+1}, \sigma_{i+1} \leftarrow$ refit Gaussian on elite set
end
$\theta = \mu$
Return $\pi_\theta(a|s)$
```

Part IV
Appendix: Supervised learning

This appendix quickly recaps some key aspects of supervised learning. Supervised learning is the workhorse under reinforcement learning, and you need to properly understand it to be able to, for example, use deep learning in RL.

12 Representing a function

We often need to represent a function (in computer memory). We are interested in representing an arbitrary function

$$f : \mathcal{X} \to \mathcal{Y},$$

where the domain and range can be discrete or continuous sets with arbitrary dimensionality. Often, we are rather interested in a conditional probability distribution, i.e., that maps

$$\mathcal{X} \to p(\mathcal{Y}),$$

where we map the domain to a probability distribution over the range. Representing a conditional probability allows us to model functions for which the input does not always give the same output (which often occurs in the real world).

12.1 Analytically known versus learned function

Analytically known function In some cases, we may represent a function in memory through an exactly known analytical expression. This is usually the case for the rules of physics, or when we make explicit assumption for a particular system.

Example: Newton's second Law of Motion states that for objects with constant mass

$$F = m \cdot a,$$

where F denotes the net force on the object, m denotes its mass, and a denotes its acceleration. In this case, the analytic expression defines the entire function, for every possible combination of the inputs.

Learned function However, for most functions in the real world, we do not know an analytical expression. Here, we enter the realm of machine learning, in particular *supervised learning*. When we do not know an analytical expression for a function, our best bet is to collect data (examples of (x, y)) and *learn the function* based on this data.

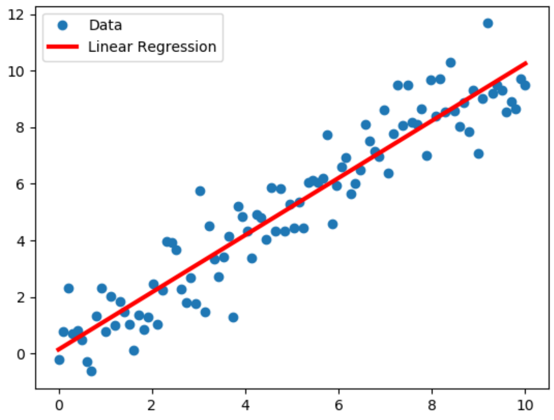


Figure 7: Example of learned function. We observe the blue datapoints. An example learned function is the red line, which properly fits the data. The red line allows us to make predictions for any new input.

Example: A company want to predict the chance that you buy a product based on your age and gender. They collect many data points of $x \in \mathbb{N} \times \{0, 1\}$, i.e., your age (a natural number) and gender (a binary indicator), that maps to $y \in \{0, 1\}$, i.e., a binary indicator whether you bought a product. They then want to *learn* the mapping

$$y = f(x).$$

12.2 Parametric versus non-parametric supervised learning

The major distinction within supervised learning is between parametric and non-parametric learning methods.

Non-parametric learning

Non-parametric learning methods summarize the function by storing the entire dataset.

Parametric methods have received most attention in literature, although non-parametric methods, like Gaussian Processes, can provide good uncertainty estimates. However, we **will focus on parametric approximation methods in this chapter**.

Parametric learning

Parametric learning methods summarize the function with a set of parameters of fixed size, i.e., independent of the number of data points.

We will generally denote the set of parameters by $\theta \in \Theta$. Often, the parameter space is continuous, i.e., $\Theta \in \mathbb{R}^{D_\theta}$. The main idea of parametric approximation methods is to first specify a function form, which takes in the observed data $x \in \mathcal{X}$ and parameters $\theta \in \Theta$. This functional form can then output

- A deterministic function, i.e.,

  $$f : \mathcal{X} \times \Theta \to \mathcal{Y},$$

  for which we will write $f_\theta(x)$ to indicate the dependence on parameters θ.

- A probability distribution, i.e.,

  $$\mathcal{X} \times \Theta \to p(\mathcal{Y}),$$

  for which we will write $p_\theta(y|x)$ to indicate the dependence on θ.

While the data (x, y) is given, the parameters (θ) are free. The key idea of parametric supervised learning is to find the parameters which best fit the data.

12.3 Key considerations of parametric supervised learning

The key steps (and considerations of parametric supervised learning are: 1) a model specification for $p_\theta(x)$, which tells us when a prediction is correct 2) an optimization criterion, i.e., some indication of when predictions are correct (better known as loss or fitness function) and 3) an optimization method, i.e., a way to find the parameter combination that gives the best performance.

1. **Model specification**: Tabular learning versus function approximation
   - **Validity**: local versus global (in principle only relevant for tabular learning)
2. **Optimization criterion**: Loss (regression versus classification) or fitness
3. **Optimization**: gradient-based or gradient-free

We will discuss these steps in the following sections.

13 Model specification

We will first discuss how to actually specify a model, without discussing yet how to optimize it. There are two main parts of model specification:

1. Model type: tabular or function approximation, which mostly depends on the dimensionality of the input x.

2. Model output: regression or classification, which depends on the space of the output y (continuous or discrete).

13.1 Model type: tabular versus function approximation

13.1.1 Tabular learning

Tabular learning methods simply store a unique output y (or distribution $p(y)$) for every input x. The parameters θ are all the individual entries in the table. Tabular methods do not actually need to be stored as a table. For example, a search tree consisting of nodes with information is in machine learning terminology also a table. Although the search tree is stored in nodes with pointers, in the background we still have a unique prediction for every input (node), i.e., a tabular format.

When the input spaces is discrete, tables are very natural. When it is continuous and we still want to use a table, we can use *discretization.* Discretization is a simple process where we divide the continuous space into bins, and only store the solution for the entire bin.

Example: Imagine we instead have data from continuous $\mathcal{X} = [0, 10]$ to discrete $\mathcal{Y} = \{0, 1\}$, but we want to store it as a table.
We can then discritize $\mathcal{X}$ into 10 bins $\{b_1, ...b_10\}$, where $b_1 = [0, 1)$, $b_2 = [1, 2)$, etc. We can then store a function $\mathcal{X} \rightarrow p(\mathcal{Y}$, e.g.,

x	**y**
$[0, 1)$	0
$[1, 2)$	0.3
$[2, 3)$	0.8
etc.	..

Local versus global tables Tabular models can be local or global.

- Learning algorithms usually approximate a *global* solution, meaning a solution for every possible input.
- We can also decide to temporarily only approximate a *local solution*, for example because we know that the global solution is too large to fit into memory. An example are planning methods, like $\alpha - \beta$ or Monte Carlo Tree Search. These methods collect data for a solution in a tree, but the tree only contains those states which are actually reached in the lookahead, not all possible states. A local solution gets discarded after a while, for example when we terminate the search (after a fixed budget) and execute the first action.

The term 'learning' is often reserved for global solution methods, i.e., methods that map an entire input space $\mathcal{X}$ to the output space. Tree search methods are therefore often not considered learning, since they use a local solution method.

13.1.2 Parametric function approximation

Parametric function approximation methods do not build an entire table for every possible point in the input domain $\mathcal{X}$. Instead, they specify a functional form, i.e., an analytic expression on how to connect *the values that* x takes to the output. This is the crucial difference. Tabular methods treat every point in the input space as unique (atomic), an never look at its actual content. Function approximation methods do look inside the actual values of the input. Thereby, they can *generalize.*

Example: We want to learn a mapping between $\mathcal{X} \in \mathbb{R}$ to $\mathcal{Y} \in \mathbb{R}$. We specify parameters $\theta = \{b, c\} \in \mathbb{R}^2$, and specify the functional form for f_θ:

$$f_\theta(x) = b + c \cdot x$$

This is a simple example of *linear regression.* We will can now try to optimize the parameters b and c to best predict the data.

The most successful class of parametric function approximators these days are (deep) neural networks. These essentially stack several layers of non-linear regressions on top of eachother, but the principle is still the same.

13.2 Comparing tabular and function approximation representation

Tables and function approximation each have their benefits and problems, which we summarize in Table 7. The benefits of tabular representations are easy to understand: 1) they are exact (for discrete input spaces), and 2) that they are relatively simple to store/learn (for example, for every input we simply store the mean output in the data). However, they also have serious drawbacks once the dimensionality of the problem increases.

However, tabular learning has two serious drawbacks: 1) they do not scale to high-dimensional problems (due to the curse of dimensionality), and 2) they cannot generalize information. We will explain both below.

13.2.1 Curse of dimensionality

So far we mostly discussed low dimensional input and output space. In real-world problems, especially the input space of functions is usually much larger.

Example: A classic problem in computer vision is image classification, i.e., predicting what type of object is visible in an image (which is also a crucial preliminary for reinforcement learning where the inputs are images). Imagine we have low-resolution greyscale images of 100x100 pixels, where each pixel takes a discrete value between 0 and 255 (a byte). Then, the input space $\mathcal{X} \in \{0, 1, .., 255\}^{100 \cdot 100}$, i.e., this space has dimensionality $100 \cdot 100 = 10.000$.

When the input space $\mathcal{X}$ is high-dimensional, we can never store the entire solution as a table. The problem is the *curse of dimensionality*:

The curse of dimensionality states that the cardinality (number of unique points) of a space scales exponentially in the dimensionality of the space.

In equations, we have that

$$|\mathcal{X}| = O\big(\exp(\mathrm{Dim}(\mathcal{X}))\big),$$

where $O(\cdot)$ ("Big-O") indicates that something "grows in the order of".

Example: Imagine have a discrete input space $\mathcal{X} = \{0,1\}^D$ that maps to a real number, $\mathcal{Y} = \mathbb{R}$, and we want to store this function as a table. We will show the required size of the table and the required memory when we use 32-bit (4 byte) floating point numbers:

Dim($\mathcal{X}$)	$\lvert\mathcal{X}\rvert$	Memory
1	2	8 Byte
5	$2^5 = 32$	128 Byte
10	$2^{10} = 1024$	4KB
20	$2^{20} \approx 1e6$	4Mb
50	$2^{50} \approx 1e15$	4.5 million TB
100	$2^{100} \approx 1e30$	5e21 TB
265	$2^{265} \approx 2e80$	-

The right column (Memory) nicely illustrates how quickly exponential growth develops. At a discrete space of size 20, it still seems like we are doing alright, storing 4 Megabyte of information. However, at a size of 50, we suddenly need to store *4.5 million Terabyte*. We can hardly imagine the numbers that follow. At an input dimensionality of size 265, our table would have grown to size $2e80$, which is roughly the estimated number of atoms in the entire universe.

Due to the curse of dimensionality, the table that we need to store to represent a function increases quickly when the size of the input space increases.

13.2.2 Generalization

A second problem of tables is that they do not generalize information.

Generalization means that - for functions in the real world - near similar input in general leads to near similar output.

We say that real-world functions are generally *smooth*. This is due to the regularity of the world, and therefore also applies to the decision-making problems studied in reinforcement learning. For example, when we need to make a right turn in our car at a certain crossing, then we will still need to make the same turn when we slightly alter the lightning.

However, tables cannot generalize, since they store an individual entry for every unique input/output pair. Therefore, the data for one pair can never influence the predictions for another pair. As we have just discussed, the tables may quickly grow really large, and we will therefore have many table entries for

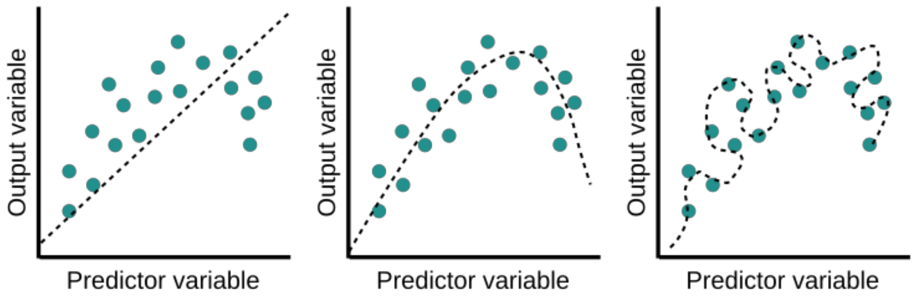


Figure 8: Illustration of generalization. Datapoints in dots, learned function as a dotted line. Middle: good generalization as present in the true function. Similar input generally has similar output (the function does not acutely fluctuate). Left: underfitting, where we lose our ability to properly generalize to new datapoints. Right: overfitting, where we also lose our ability to make accurate predictions for new datapoints.

Table 7: Benefits and problems of tabular representation versus function approximation

	Benefit	**Problem**
Table	• Exact • (Easy to design)	• Curse of dimensionality • No generalization
Function approximation	• Generalization • Lower memory requirement (scales to high-dim)	• Approximation errors

which we have not observed any data. Therefore, generalization is crucial in large problems.[3]

In short, both tabular representations and function approximation have their benefits, summarized in Table 7. In large problems, we will in principle always need function approximation, if we want to store a global solution (i.e., for every possible input). However, tabular approaches have their value in small problems, and also in larger problems when we use local solution, e.g., only store a solution for a small number of input points of current interest.

[3]Note that discretization of a continuous space does allow for partial generalization within a bin. However, we will fit a stepwise function over the bins, which is an approximation class with very low capacity (we can not represent many functions), and we will likely not generalize well.

13.3 Output type: regression versus classification

The final part of the model specification depends on the type of output space. When the output space is continuous, we call it a regression problem, while a discrete output space relates to classification.

13.3.1 Regression

Deterministic continuous prediction When the output space is continuous and we want a deterministic prediction, we can simply output a vector of number with the same dimensionality as y:

$$y = f_\theta(x)$$

Continuous distribution When we instead want to predict an entire distribution, we can let the model output parameters of a continuous probability distribution. For example, we could assume a Normal distribution for y, let $f_\theta(x)$ output two numbers for every x, and specify:

$$p_\theta(y|x) = \mathcal{N}(\mu_\theta(x), \sigma_\theta(x)) = \frac{1}{\sigma_\theta(x)\sqrt{2\pi}} \exp\Big(- \frac{(y - \mu_\theta(x))^2}{2\sigma_\theta(x)^2}\Big)$$

where the mean $\mu_\theta(x) = f_{\theta,1}(x)$ is the first element of the model output, and the standard deviation $\sigma_\theta(x) = e^{f_{\theta,2}(x)}$ is the exponential of the second element. The exponentiation ensures that the variance stays positive.

13.3.2 Classification

For classification we will always output a discrete distribution (there is no deterministic discrete prediction). To specify the parameters of a discrete distribution of size K, we need to predict K elements (or actually $K-1$) and ensure that they sum to one. The common way to ensure this is through the *softmax function*. We let the model output a vector $f_\theta(x)$ of length K, and specify the discrete distribution as:

$$p_\theta(y|x) = \text{softmax}(f_\theta(x)) = \frac{e^{f_\theta(x)}}{\sum_k e^{f_{\theta,k}(x)}}$$

where the denominator sums over the elements of $f_\theta(x)$. The exponentiation again ensures that probabilities are positive, and the denominator term ensures that they sum to one. Often, we also introduce a temperature parameter τ.

14 Loss function

To find the best parameters, we typically need some indication when the learned function performs well. We can do this by specifying a loss function. The loss should penalize when our prediction is far off from the true observation. We can in principle come up with any loss function we want, as long as it penalizes the model when it makes predictions far away from the data. A simple example is the well-known mean-squared error loss for regression problems:

$$\mathrm{L}^{MSE}(\theta|\mathcal{D}) = \frac{1}{N_{\mathcal{D}}}\sum_{i=1}^{N_{\mathcal{D}}}(y_i - f_\theta(x_i))^2$$

where $\mathcal{D}$ denotes the dataset, and i indexes over the dataset. Squaring the error between each prediction and true observation ensures that all losses are positive, i.e., negative and positive errors have equal contribution.

It turns out that the MSE loss is actually a special case of a more generic principle, known as maximum likelihood estimation (MLE). We have already seen that it can be useful to predict the entire distribution $p_\theta(y|x)$, since it allows us to model stochastic phenomena. However, it turns out that we can also use these probibalistic approaches to construct loss function, by finding parameters that maximize the probability of the observed data.

14.1 Maximum likelihood estimation

A common approach to loss functions is to use *maximum likelihood estimation*, which attempts to find the parameters that maximize the probability of the observed data. In those cases, we are actually learning a conditional probability distribution $p : \mathcal{X} \times \Theta \to p(\mathcal{Y})$.

The likelihood of the data is

$$p_\theta(y|x) = \prod_{i=1}^{N_{\mathcal{D}}} p_\theta(y_i|x_i),$$

and our goal is to find the parameters that maximize the likelihood of the data, known as the *maximum likelihood estimate* (MLE):$\theta^\star = \arg\max_\theta p_\theta(y|x)$. We can turn the likelihood into a loss ('something we want to minimize'), by taking the *negative log likelihood* (NLL):

$$
\begin{aligned}
\mathrm{NLL}(\theta|y,\mathbf{x}) &= -\log p_\theta(y|x) \\
&= -\log \prod_{i=1}^{N_{\mathcal{D}}} p_\theta(y_i|x_i) \\
&= \sum_{i=1}^{N_{\mathcal{D}}} -\log p_\theta(y_i|x_i) \\
&\sim \frac{1}{N_{\mathcal{D}}} \sum_{i=1}^{N_{\mathcal{D}}} -\log p_\theta(y_i|x_i) \\
&= \mathbb{E}_{\mathcal{D}}\Bigg[-\log p_\theta(y|x) \Bigg] \qquad (38)
\end{aligned}
$$

which we can do because the log function is monotone, and it also nicely turns the product into a sum. The last equation is actually the **cross-entropy** between the data distribution and the model distribution:

$$H[p_{\mathrm{data}}(y|x), p_\theta(y|x)].$$

See Chapter 2 for details on the cross-entropy. We will now show the two most common applications of this idea, for regression (continuous y) and classification (discrete y).

14.2 Regression

Regression settings are commonly trained on the *mean-squared error* (MSE). We will show how this is actually a form of maximum likelihood estimation when we assume a Gaussian distribution with fixed standard deviation. We assume the following form for $p(y|x)$:

$$p(y|x) = \mathcal{N}(f_\theta(x), \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\Big(-\frac{(y - f_\theta(x))^2}{2\sigma^2}\Big)$$

i.e., a normal distribution where we only predict the mean as $f_\theta(x)$ and fix the standard deviation at some arbitrary constant a.

When we plug the above equation into the NLL equation, and drop the dependence on σ (since it is a constant we will not optimize for), we get:

$$
\begin{aligned}
\mathrm{L}(\theta|y,x) &= \frac{1}{N_{\mathcal{D}}}\sum_{i=1}^{N_{\mathcal{D}}} -\log p_\theta(y_i|x_i) \\
\mathrm{L}(\theta|y,x) &= \frac{1}{N_{\mathcal{D}}}\sum_{i=1}^{N_{\mathcal{D}}} -\log\Big(\frac{1}{\sigma\sqrt{2\pi}}\exp\Big(-\frac{(y_i-f_\theta(x_i))^2}{2\sigma^2}\Big)\Big) \\
\mathrm{L}(\theta|y,x) &= \frac{1}{N_{\mathcal{D}}}\sum_{i=1}^{N_{\mathcal{D}}}(y_i-f_\theta(x_i))^2 \qquad (39)
\end{aligned}
$$

where $\mathcal{D}$ denotes the dataset of size $N_{\mathcal{D}}$, and i indexes over entries in the dataset. This is known as the *mean-squared error* (MSE) loss, and is very common in supervised learning. In effect, we simply take the error between the true output y_i and our prediction $f_\theta(x_i)$, which is very intuitive. The squaring of the error ensures that we penalize both positive and negative errors (in equal amount). Note that this a function of θ, since we want to optimize these parameters.

Example: For the model $y = f_\theta(x) = b + c\cdot x$, the MSE loss would be

$$\mathcal{L}(\theta|y,x) = \frac{1}{N_{\mathcal{D}}}\sum_{i=1}^{N_{\mathcal{D}}}\Big(b + c\cdot x_i - y_i\Big)^2,$$

$$y = f_\theta(x) = b + c\cdot x$$

14.3 Classification

Classification problems, where the output is discrete, are often trained on the *cross-entropy* loss with, for example, softmax predictions of $p_\theta(y|x)$:

$$
\begin{aligned}
\mathcal{L}(\theta|y,x) &== \mathbb{E}_{\mathcal{D}}\Bigg[-\log p_\theta(y|x)\Bigg] \\
&= -\frac{1}{N_{\mathcal{D}}}\sum_{i=1}^{N_{\mathcal{D}}}\log\Big(\frac{e^{f_{\theta,y_i}(x)}}{\sum_k e^{f_{\theta,k}(x)}}\Big) \qquad (40)
\end{aligned}
$$

where $f_{\theta,y_i}(x)$ means that we select the element of $f_\theta(x)$ for which y_i is equal to 1 (i.e., the true label). The above equation essentially maximizes the probability of each true observed label.

Example: Imagine we have a three class classification problem with a single independent variable, i.e., $\mathcal{X} = \mathbb{R}$ and $\mathcal{Y} = \{0,1\}^3$ (we call this a *one-hot encoding*, where we have a vector for the number of categories, which has a single one at the true observation). We will assume the parameters vectors $\boldsymbol{b} \in \mathbb{R}^3$ and $\boldsymbol{c} \in \mathbb{R}^{\mathbf{3}}$ are vectors of size three. We specify:

$$f_\theta(x) = \boldsymbol{b} + \boldsymbol{c} \cdot x$$

which now also outputs a vector of length three.
We specify $p_\theta(y|x) = \text{softmax}(f_\theta(x))$ which outputs a probability distribution of length three. When we use a one-hot encoding for the labels, $\boldsymbol{y} \in [0,1]^3$, we can train our model on

$$\mathcal{L}(\theta|y,x) = -\frac{1}{N_{\mathcal{D}}} \sum_{i=1}^{N_{\mathcal{D}}} \boldsymbol{y}_i \cdot \log\left(\frac{e^{\boldsymbol{b}+\boldsymbol{c}\cdot x_i}}{\sum_k e^{\boldsymbol{b}+\boldsymbol{c}\cdot x_{i,k}}}\right)$$

This model is better known as *logistic regression*.

15 Optimization

In the last step we want to find the optimal parameters $\theta^\star$, i.e., the parameters which make the loss on the observed data small (or actually, the parameters which make the loss on a validation set small).

$$\theta^\star = \arg\min_\theta L(\theta|\mathcal{D}) \tag{41}$$

15.1 Gradient-based optimization

Gradient-based optimization use the derivative of the objective to find the optimum. In the case of a loss minimization, we may then apply *gradient descent*, which in each iteration of the algorithm pushes the parameters in the direction of the negative gradient:

$$\theta' = \theta - \eta \cdot \frac{\partial L(\theta|\mathcal{D})}{\partial \theta} \tag{42}$$

for learning rate $\eta \in \mathbb{R}^+$. A full algorithm for gradient *ascent*, which only changes the minus into a plus, is given in Alg. 2.

15.1.1 Gradient-based updates on tables

In a deterministic table, the approximation function for a datapoint x is actually a tabular look-up

$$f_\theta(x) = \theta_x, \tag{43}$$

i.e., we search for the table entry θ_j where the datapoint x belongs to.

Maximum likelihood on tables In the tabular approach, we may actually analytically find the optimum of the above optimization. For a single table entry we write θ_j. We will show the case for a regression, where the table entry contains a continuous number, where we use a mean-squared error:

$$L(\theta|\mathcal{D}) = \sum_{j=1}^{n_\theta} \sum_{k=1}^{n_j} \frac{1}{2}(\theta_j - y_{j,k})^2$$

where j sums over all parameters, and k sums over all entries in the data where $x_{j,k}$ belongs to table entry θ_j. We can differentiate the above expression towards a single table parameter:

$$\frac{\partial L(\theta|\mathcal{D})}{\partial \theta_j} = \sum_{k=1}^{n_j} (\theta_j - y_{j,k}) \tag{44}$$

When we equate the above expression to 0, we can solve for θ_j:

$$\theta_j = \frac{\sum_{k=1}^{n_j} y_{j,k}}{n_j}. \tag{45}$$

This is a very intuitive result, since we simply set each table entry to the mean of the observations for that cell. We call it the *tabular maximum likelihood estimate.*

Example: We learn a function from $\mathcal{X} = \{1, 2, 3\}$ to discrete $\mathcal{Y} = \{0, 1\}$. We observe $D = \langle(1, 0), (2, 0), (2, 1), (1, 0), (1, 1), (3, 0), (3, 0)\rangle$. The tabular model would estimate (verify this):

x	**y**
1	1/3
2	1/2
3	0

Incremental/general tabular updates Often, we do not want to update a table on an entire dataset, but rather have data as an incoming stream, for example during planning or RL. We no longer find an exact solution, but rather gradually move the solution in the direction of the observed data. We can follow the general learning scheme of Eq. 42 for a table to get an incremental learning scheme. For simplicity, we will write the equation for the update of a single table entry θ_j based on an observed data-pair (x, y), where x_i falls in the table entry of θ_j. Let's assume a mean-squared error loss, for which we already computed the gradient in Eq. 44. Plugging this equation into Eq. 42, we get the generic tabular update:

$$\theta_j \leftarrow \theta_j - \eta \cdot (\theta_j - y_{j,k}) \tag{46}$$

Rearranging terms we get:

$$\theta_j \leftarrow (1 - \eta) \cdot \theta_j + \eta \cdot y_{j,k} \tag{47}$$

where $\eta \in [0, 1)$, i.e., in the case of tabular learning the learning rate has a restriction. The above formulation is the generic tabular learning update, which moves the estimate a small step in the direction of the new observed data. Some special cases of this equation are:

- Standard learning update: Where $\eta \in (0,1)$ is fixed throughout learning.
- Replace update: When we set $\eta = 1$, we get

$$\theta_j \leftarrow y_{j,k}$$

  , which completely replaces the new table entry with the current estimate at every step. We observe this in some planning approaches, like A* or α-β pruning.
- Average/MLE update: When we set $\eta = \frac{1}{n}$, we get

$$\theta_j \leftarrow \frac{n-1}{n} \cdot \theta_j + \frac{1}{n} \cdot y_{j,k},$$

  where n denotes the number of updates so far. This essentially retrieves the MLE estimate (Eq. 45), but in an incremental update fashion. This averaging update is for example used in Monte Carlo Tree Search (MCTS).

Note that these planning methods are technically not considered learning, but as we see one can phrase their updates as tabular learning updates. The above discussion of tabular updates may seem a bit lengthy and complicated for such intuitive choices. Nevertheless, it is good to see the bigger picture: how tabular updates are based on the same principle as function approximation updates, and how planning updates are actually tabular representation methods.

15.1.2 Gradient-based updates with function approximation

Gradient-based optimization on function approximation is usually implemented with gradient descent and the chain rule. We will take deterministic regression as an example. We want to compute the gradient

$$\begin{aligned}\nabla_\theta \mathcal{L}(\theta|\mathcal{D}) &= \nabla_\theta \mathbb{E}_{\mathcal{D}}\Big[\Big(f_\theta(x) - y_i\Big)^2\Big] \\ &= \frac{1}{N_{\mathcal{D}}} \sum_{i=1}^{N_{\mathcal{D}}} \nabla_\theta \Big(f_\theta(x) - y_i\Big)^2 \end{aligned} \tag{48}$$

However, when the size N of the dataset $\mathcal{D}$ grows larger, then the above summation becomes very computationally expensive. Therefore, most practical algorithms *approximate* the above gradient with a much smaller sample from the dataset, a **minibatch** of size M(common choices of M are 32 or 64). We call this *stochastic gradient descent* (SGD). The SGD gradient becomes:

$$\nabla_\theta \mathcal{L}(\theta|\mathcal{D}) \approx \frac{N_\mathcal{D}}{M} \sum_{m=1}^{M} \nabla_\theta \Big(f_\theta(x_m) - y_m \Big)^2$$

It turns out that this algorithms works very well in practice. The gradient is noisy, but often reaches a reasonable optimum. The main benefit of the algorithm is that it scales to large datasets (as we can keep M fixed to a small number, independent of dataset size $N_\mathcal{D}$).

15.2 Gradient-free optimization

Finally, we can also use gradient-free optimization methods, although gradient-based methods largely dominate the supervised learning community at this moment. Examples include evolutionary strategies, simulated annealing, the cross-entropy method, etc. We will not go into details here.