Fundamentals of Deep Reinforcement Learning

Pedro Santana
Iscte – University Institute of Lisbon

September 16, 2025

Abstract

This document serves as an introduction to deep reinforcement learning, specifically designed for students with a college-level background in mathematics and computer science. While such pre-existing knowledge is expected, the document includes a mathematical primer to recall the basic principles necessary for understanding reinforcement learning algorithms. Adopting a concise, depth-first approach to the literature, the document introduces the formalism of Markov Decision Processes (MDPs) and presents both exact and approximate methods to solve them. For approximate methods, it covers the classical REINFORCE algorithm and then delves into Proximal Policy Optimization (PPO), one of the most widely used and effective reinforcement learning algorithms for controlling embodied agents.

The document concludes with a list of references that served as basis for its preparation. Additionally, they are recommended complementary material for students eager to further their understanding of deep reinforcement learning. For instance, Stable Baselines3 (SB3) [6] provides reliable implementations of reinforcement learning algorithms in PyTorch, making it an excellent tool for experimentation. Additionally, educational resources from OpenAI [5] offer an interesting blend of theory and practice. The textbook by Sutton and Barto [8] is also a highly recommended resource, offering a comprehensive introduction to reinforcement learning concepts and techniques.

Releases

- 2025/09/16: Rev. 20250916
- 2025/05/05: Rev. 20250505
- 2025/03/10: Rev. 20250310

Citation
Santana, P. (2025) Fundamentals of Deep Reinforcement Learning, Rev. 20250916, Iscte-IUL.

Contents

1 Math Primer

1.1 Probabilities

When we perform some kind of experiment in the real-world we should always take into account the possibility of randomness. Although we are not fully sure of the outcome of the experiment, we often know the set of all possible outcomes. To the set of all possible outcomes of an experiment we call *sample space*, Ω. For instance, in an experiment consisting of tossing fair coins, then $\Omega = \{H, T\}$, where H and T mean that the outcome of the toss is head and tail, respectively. If the experiment is tossing two fair coins, then its sample space is $\Omega = \{(HH), (HT), (TH), (TT)\}$.

Any subset E of the sample space Ω, $E \subset \Omega$, is known as an *event*. For instance, $E = \{(H, H), (H, T)\}$ refers to the event that a head appears on the first coin. The probability of every event E of the sample space Ω to occur is $P(E)$, which is subject to the following conditions:

$$0 \leq P(E) \leq 1 \quad \text{and} \quad P(\Omega) = \sum_{o \in \Omega} P(o) = 1.$$

For example, if we toss two fair coins, the probability of getting two heads is $P(\{HH\}) = \frac{1}{4}$, given that the sample space has four equally probable elements. In the same experiment, the probability of getting a single head is $P(\{HT, TH\}) = P(\{HT\}) + P(\{TH\}) = \frac{1}{4} + \frac{1}{4} = \frac{1}{2}$.

1.2 Discrete Random Variables

We may be interested on certain functions of experiment outcomes rather than the outcomes themselves. For instance, when rolling two dice, we might be more interested in the sum of the dice than the specific numbers rolled. We might want to know that the sum is seven, without caring whether the roll was (1, 6), (2, 5), (3, 4), (4, 3), (5, 2), or (6, 1). These functions of outcomes are known as *random variables*. They are called random because their specific values are determined by chance.

Formally, a *discrete random variable* X is a deterministic function that assigns a real number to each outcome in a sample space of a random experiment, $X : \Omega \to \mathbb{R}$, where the random variable takes on a countable number of distinct values. The complete set of values that the random variable X can take is denoted by $R(X)$.

The probability that a discrete random variable X takes a specific value $x \in R(X)$ is defined by its Probability Mass Function (PMF), denoted as $P(X = x)$. For simplicity, $P(X = x)$ will be referred to hereafter simply as the probability of X taking the value x.

For the sake of simplicity, instead of $\sum_{x \in R(X)}$, we use $\sum_x$ to denote a summation over all possible values that the random variable X can take. Following the definition of probability, the following conditions must hold:

$$0 \leq P(X = x) \leq 1 \quad \text{and} \quad \sum_x P(X = x) = 1.$$

Let us reanalyze the simple example in which we toss a fair coin twice. In this experiment, whose sample space is $\Omega = \{HH, HT, TH, TT\}$. Let the discrete random variable X be the number of heads we observe in these two coin tosses. Therefore, X can take the values 0, 1, or 2:

$$X(TT) = 0, X(HT) = 1, X(TH) = 1, X(HH) = 2.$$

Note that this example clearly shows that the random variable X is a simple deterministic function that maps an occurred outcome to a real-valued scalar. The randomness is fully contained in the experiment, that is, in determining which outcome actually occurs. This approach allows to decouple the randomness of the experiment from its deterministic mathematical treatment. Let us get back to our example. Since the coin is fair, each of the four outcomes is equally likely and, thus,

$$\begin{aligned} P(X = 0) &= P(\{TT\}) = \frac{1}{4}, \\ P(X = 1) &= P(\{HT, TH\}) = \frac{1}{2}, \\ P(X = 2) &= P(\{HH\}) = \frac{1}{4}. \end{aligned}$$

Note that the probabilities sum up to 1, satisfying the condition for probability distributions,

$$\sum_x P(X = x) = P(X = 0) + P(X = 1) + P(X = 2) = \frac{1}{4} + \frac{1}{2} + \frac{1}{4} = 1.$$

1.3 Continuous Random Variables

In some experiments, we may be interested in outcomes that vary over a continuous range rather than distinct discrete values. In these cases, the random variable is called of *continuous random variables* as it takes one of an uncountable number of distinct values.

Formally, a *continuous random variable* X is a deterministic function that assigns a real number to each outcome in a sample space of a random experiment, $X : \Omega \to \mathbb{R}$. Unlike discrete random variables, a continuous random variable can take any value within a certain range or interval. The probability that X takes a specific value is always zero, as there are infinitely many possible values in any continuous range. Instead, probabilities are defined over intervals and are determined using a *probability density function (PDF)*, denoted by $f_X(x)$. The function $f_X(x)$ satisfies the following conditions:

$$f_X(x) \geq 0 \quad \text{for all } x \in \mathbb{R},$$
$$\int_{-\infty}^{\infty} f_X(x)\, dx = 1.$$

The probability that X falls within an interval $[a, b]$ is given by the integral of the PDF over that interval:

$$P(a \leq X \leq b) = \int_a^b f_X(x)\, dx.$$

Uniform Distribution

Suppose X represents the random outcome of selecting a point uniformly at random on the interval $[0, 1]$. In this case, the PDF of X is defined as follows:

$$f_X(x) = \begin{cases} 1 & \text{if } 0 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases},$$

which is a valid PDF because $f_X(x) \geq 0$ for all x and it integrates to 1 over the entire domain (recall that the primitive of 1 is x):

$$\int_{-\infty}^{\infty} f_X(x)\, dx = \int_0^1 1\, dx = [x]_0^1 = 1 - 0 = 1.$$

The probability that X lies within a subinterval $[a, b]$, where $0 \leq a < b \leq 1$, is:

$$P(a \leq X \leq b) = \int_a^b f_X(x)\, dx = \int_a^b 1\, dx = [x]_a^b = b - a.$$

For example, the probability that X lies in $[0.25, 0.75]$ is:

$$P(0.25 \leq X \leq 0.75) = \int_{0.25}^{0.75} 1\, dx = [x]_{0.25}^{0.75} = 0.75 - 0.25 = 0.5.$$

Since the distribution is uniform over $[0, 1]$, each subinterval of equal length has the same probability. The length of the interval $[0.25, 0.75]$ is $0.75 - 0.25 = 0.5$, which is half the length of the entire interval $[0, 1]$. Therefore, the probability of X falling within this subinterval is exactly 0.5, as we have obtained.

Gaussian Distribution

Let us now analyze the case of a random variable X that follows a Gaussian (normal) distribution with mean μ and standard deviation σ. Such a distribution is denoted by $\mathcal{N}(\mu, \sigma^2)$, where σ^2 is the variance. The mean defines the center of the distribution, which is the most probable outcome, while the standard deviation defines the spread or dispersion of the values around the mean. Figure 1 illustrates the PDFs of Gaussian distributions with different means and standard deviations.

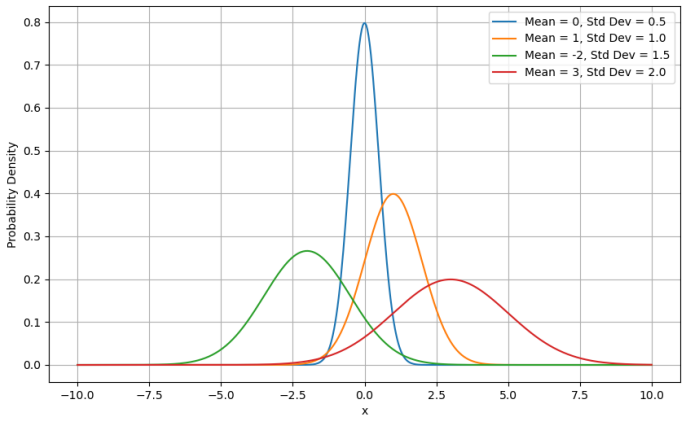


Figure 1: PDFs of Gaussian distributions with different means and standard deviations.

Formally, the PDF of a continuous random variable X that follows a Gaussian distribution with mean μ and standard deviation σ, is defined as:

$$f_X(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right). \tag{1}$$

1.4 Sampling Distributions

Sampling is the process of obtaining a specific value of a random variable according to a given probability distribution. This process is represented using the symbol $\sim$. For example, obtaining a sample x from a Gaussian probability density function (PDF) is denoted as:

$$x \sim \mathcal{N}(\mu, \sigma^2).$$

The distribution, whether Gaussian or not, can arise naturally in real-world phenomena. In such cases, sampling is performed by measuring some quantity, such as the current temperature at your location. Alternatively, distributions can be represented computationally, where samples are generated programmatically. In computational sampling, a procedure is used to randomly select a value based on the probability mass function (PMF) for discrete random variables or the probability density function (PDF) for continuous random variables.

1.5 Joint Probability of Discrete Random Variables

For discrete random variables X and Y, the joint probability $P(X = x, Y = y)$ represents the probability that X takes the value x and Y takes the value y simultaneously. This can be viewed as the probability of the event where both conditions $X = x$ and $Y = y$ are satisfied at the same time. Formally, it is expressed as

$$P(X = x, Y = y).$$

The sum of the joint probabilities over all possible pairs of values (x, y) must be 1, as it encompasses all possible outcomes of the random variables X and Y. This is a requirement for the joint probability distribution to be valid. Mathematically, this is represented as

$$\sum_x \sum_y P(X = x, Y = y) = 1.$$

If the random variables X and Y are independent, the joint probability is the product of the individual probabilities,

$$P(X = x, Y = y) = P(X = x) \cdot P(Y = y).$$

Example: ChatBot

Let us imagine a study where we surveyed 100 potential users to determine their preferences for interacting with a ChatBot that has a cartoonish appearance. The objective of the study was to analyze whether these users would like or dislike such a ChatBot and how their preferences are distributed between children and adults. Assuming the sample size is representative of the overall population, the statistics derived from this survey can be generalized.

Table 1 shows the distribution of users based on their age group and preferences. For example, 30 individuals are children who like the cartoonish character. By dividing each number in the table by the total sample size (100), we can calculate the joint probability distribution over the random variables X and Y, denoted as $P(X = x, Y = y)$, where $x \in \{\text{like}, \text{dislike}\}$ and $y \in \{\text{child}, \text{adult}\}$. For instance, the 30 children who like the cartoonish character represent a joint probability of $P(X = \text{like}, Y = \text{child}) = 0.3$, which can be interpreted as 30%. Table 2 provides the complete joint probability distribution. Please remember that this is a hypothetical survey (no actual user study was conducted) and most likely it does not accurately reflect real-world preferences.

	Like	**Dislike**	Total
Child	30	10	40
Adult	20	40	60
Total	50	50	100

Table 1: Distribution of user preferences for the ChatBot example.

	Like	**Dislike**
Child	$P(X = \text{like}, Y = \text{child}) = 0.3$	$P(X = \text{dislike}, Y = \text{child}) = 0.1$
Adult	$P(X = \text{like}, Y = \text{adult}) = 0.2$	$P(X = \text{dislike}, Y = \text{adult}) = 0.4$

Table 2: Joint probability distribution for the ChatBot example.

1.6 The Log Trick

When dealing with a sequence of events for time steps $t = 1, 2, \ldots, n$ for a set of independent random variables $X_1, X_2, \ldots, X_n$, the joint probability of these events occurring is given by

$$P(X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n) = \prod_{i=1}^{n} P(X_i = x_i),$$

which represents the probability of observing the specific values $x_1, x_2, \ldots, x_n$ for the corresponding random variables $X_1, X_2, \ldots, X_n$.

To avoid the numerical instability that can arise from multiplying several very small probabilities, we often use the log trick. Taking the logarithm of the joint probability transforms the product of probabilities into a sum of logarithms, as a consequence of the logarithmic identity $\log(xy) = \log(x) + \log(y)$. This transformation simplifies calculations and enhances numerical stability, as it is easier to handle sums of logarithms than products of small numbers:

$$\log P(X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n) = \log \left(\prod_{i=1}^{n} P(X_i = x_i) \right) = \sum_{i=1}^{n} \log P(X_i = x_i).$$

In optimization problems, the log trick is particularly useful. Optimizing the log probability yields the same result as optimizing the probability itself because the logarithm is a monotonically increasing function. This means that the values of the probability and the log probability increase and decrease together. Therefore, finding the maximum of the log probability will also find the maximum of the probability. This simplifies the optimization process and often makes it more computationally efficient.

1.7 Probability Marginalization

Marginalization is the process of summing the joint probability distribution over one of the variables to obtain the marginal probability distribution of the other variable. This process gives the probabilities

of each variable individually, without reference to the other variable. For instance, to find the marginal probability of X, we sum over all possible values of Y:

$$P(X = x) = \sum_y P(X = x, Y = y).$$

Similarly, to find the marginal probability of Y, we sum over all possible values of X:

$$P(Y = y) = \sum_x P(X = x, Y = y).$$

In these equations, $P(X = x)$ is the probability that X takes the value x regardless of the value of Y while $P(Y = y)$ is the probability that Y takes the value y regardless of the value of X.

Example: ChatBot (continued)

Table 3 illustrates both the joint and marginal probability distributions for the ChatBot example. Recall that this hypothetical survey examines user preferences for a ChatBot with a cartoonish appearance, segmented by age groups (children and adults) and their likes or dislikes.

The marginal probabilities are derived by summing the joint probabilities across the different categories of the other variable. For instance, the marginal probability $P(X = \text{child}) = 0.4$ is obtained by adding $P(X = \text{child}, Y = \text{like})$ and $P(X = \text{child}, Y = \text{dislike})$, which represent the total probability of being a child regardless of preference. The probability of liking the ChatBot regardlessness of age, $P(Y = \text{like}) = 0.5$, is found by summing the probabilities of both children and adults who like it.

	Like	**Dislike**	Total
Child	$P(X = \text{child}, Y = \text{like}) = 0.3$	$P(X = \text{child}, Y = \text{dislike}) = 0.1$	$P(X = \text{child}) = 0.4$
Adult	$P(X = \text{adult}, Y = \text{like}) = 0.2$	$P(X = \text{adult}, Y = \text{dislike}) = 0.4$	$P(X = \text{adult}) = 0.6$
Total	$P(Y = \text{like}) = 0.5$	$P(Y = \text{dislike}) = 0.5$	

Table 3: Joint and marginal probability distributions for the ChatBot example.

1.8 Conditional Probability

For discrete random variables X and Y, the conditional probability $P(X = x \mid Y = y)$ represents the probability that X takes the value x given that Y has taken the value y. This can be interpreted as the proportion of times X takes the value x out of the times Y takes the value y. Formally, this is defined as

$$P(X = x \mid Y = y) = \frac{P(X = x, Y = y)}{P(Y = y)}, \quad \text{provided } P(Y = y) > 0.$$

Moreover, the conditional probability $P(X = x \mid Y = y)$ itself forms a probability distribution over X for any given y. Therefore, the sum of $P(X = x \mid Y = y)$ over all possible values of x must be 1:

$$\sum_x P(X = x \mid Y = y) = 1, \quad \forall y \in \text{Range}(Y) \text{ such that } P(Y = y) > 0.$$

Example: ChatBot (continued)

Let us proceed with the analyses of our ChatBot example. Table 3 shows that the marginal probabilities for liking and disliking the ChatBot are $P(Y = \text{like}) = 0.5$ and $P(Y = \text{dislike}) = 0.5$, indicating that, overall, the probability of a user liking the ChatBot is equal to the probability of disliking it. However, examining the conditional probabilities provides additional insights. The conditional probability of a user liking the ChatBot given they are a child is:

$$P(Y = \text{like} \mid X = \text{child}) = \frac{P(X = \text{child}, Y = \text{like})}{P(X = \text{child})} = \frac{0.3}{0.4} = 0.75.$$

This can be interpreted as 75% of children liking the ChatBot. Similarly, the conditional probability of a user liking the ChatBot given they are an adult is:

$$P(Y = \text{like} \mid X = \text{adult}) = \frac{P(X = \text{adult}, Y = \text{like})}{P(X = \text{adult})} = \frac{0.2}{0.6} = 0.33(3).$$

These conditional probabilities reveal that children are more likely to like the ChatBot compared to adults. Specifically, 75% of children like the ChatBot, while only 33.3% of adults do. Note that, although the marginal probabilities indicate a higher likelihood of encountering an adult ($P(X = \text{adult}) = 0.6$) than a child ($P(X = \text{child}) = 0.4$), the conditional probabilities highlight that children have a stronger preference for the ChatBot. Thus, this example shows how different aspects of the data can be revealed through different types of probability distributions.

1.9 Chain Rule of Probability

The chain rule of probability, which follows directly from the definition of conditional probability, states how joint distributions over random variables can be decomposed into conditional distributions over only one variable:

$$\begin{aligned} P(X = x, Y = y) &= P(X = x \mid Y = y)P(Y = y) \\ &= P(Y = y \mid X = x)P(X = x) \end{aligned}$$

Via simple recursive substitutions, the rule generalizes to n random variables as follows:

$$P(X_1 = x_1, \ldots X_n = x_n) = P(X_1 = x_1) \prod_{i=2}^{n} P(X_i = x_i \mid X_1 = x_1, \ldots X_{i-1} = x_{i-1}).$$

When stating probability equations, we often consider the joint probability of groups of variables as single entities. For example, for $n = 3$, we can derive the following decompositions:

$$\begin{aligned} P(X = x, Y = y, Z = z) &= P(X = x)P(Y = y \mid X = x)P(Z = z \mid X = x, Y = y) \\ &= P(X = x \mid Y = y, Z = z)P(Y = y, Z = z) \\ &= P(X = x, Y = y \mid Z = z)P(Z = z). \end{aligned}$$

1.10 Independence and Conditional Independence

Two random variables X and Y are considered *independent* if their joint probability distribution can be expressed as the product of their individual marginal probability distributions:

$$P(X = x, Y = y) = P(X = x)P(Y = y), \quad \forall x \in R(X), y \in R(Y).$$

Note that this equation defines joint probability without referring to any conditional probability. That is, when two random variables are independent, knowing the value of one provides no information about the value of the other. This can also be expressed as follows:

$$\begin{aligned} P(X = x \mid Y = y) &= P(X = x), \quad \forall x \in R(X), y \in R(Y). \\ P(Y = y \mid X = x) &= P(Y = y), \quad \forall x \in R(X), y \in R(Y). \end{aligned}$$

Two random variables X and Y are *conditionally independent* given a random variable Z if, for every value of Z, the conditional probability distribution of X and Y can be expressed as a product of the individual conditional distributions of X given Z and X given Z:

$$\begin{aligned} &P(X = x, Y = y \mid Z = z) = \\ &\qquad P(X = x \mid Z = z)P(Y = y \mid Z = z), \quad \forall x \in R(X), y \in R(Y), z \in R(Z). \end{aligned}$$

Example: ChatBot (continued)

We can use the probabilities expressed in Table 3 to determine whether the random variables X (age group) and Y (preference) in our ChatBot example are independent. For the random variables to be independent, the joint probability must equal the product of the marginal probabilities. However, this is not the case in our example. For instance, from the table, we see that:

$$P(X = \text{child}, Y = \text{like}) \neq P(X = \text{child}) \cdot P(Y = \text{like}) \quad (\text{because } 0.3 \neq 0.4 \cdot 0.5).$$

This discrepancy indicates that knowing the user's age group provides information about their likelihood of liking or disliking the ChatBot. This conclusion is further supported by examining the conditional probability:

$$P(Y = \text{like} \mid X = \text{child}) \neq P(Y = \text{like}) \quad (\text{because } 0.75 \neq 0.5).$$

Hence, we can say that the preferences are not independent of the age group, as the likelihood of liking the ChatBot differs when conditioned on the age group.

1.11 Expected Value

The expected value, or *mean*, of a random variable provides a measure of the central tendency of the distribution of the random variable. It represents the long-term average outcome of the random variable when the process is repeated infinitely many times.

Discrete Random Variables

For a discrete random variable X, the expected value is defined as the weighted average of all possible values that X can take, with the weights being the probabilities of each value:

$$\mathbb{E}[X] = \sum_x x \cdot P(X = x).$$

For example, let us consider a random variable X that can take the values $X = 1$ and $X = 2$ with probabilities of 70% and 30%, respectively. In this case, the expected value of X is calculated as $0.7 \cdot 1 + 0.3 \cdot 2 = 1.3$. This means that, on average, the values that X takes in repeated random experiments will converge to 1.3 over time. While X may never actually take the value 1.3, this is the average outcome we can expect in the long run.

Continuous Random Variables

For a continuous random variable X, the expected value is calculated using the probability density function, $f_X(x)$, which describes the distribution of X (see Section 1.3). In this case the summation used for discrete random variables is replaced by an integral because the possible values of X are spread over a continuous range (x represents specific realizations (values) of the random variable X):

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x \cdot f_X(x)\, dx.$$

Sampling-Based Estimates

When we do not know the probabilities of each outcome, we can take n independent samples from the underlying distribution and consider the sample mean as an approximation to the distribution's true expected value. Assuming that the samples are obtained under the same distribution (i.e., its statistical properties are invariant over time), the Law of Large Numbers provides us with the guarantee that the sample mean will converge to the mean of that distribution with probability 1 as n approaches infinity.

Formally, let $X_1, X_2, \ldots$ be a sequence of independent random variables having a common distribution with mean μ, $\mathbb{E}[X_i] = \mu$. Each of these random variables takes a specific value when we sample from the underlying distribution. The law of large numbers tells us that, with probability 1,

$$\frac{X_1 + X_2 + \ldots + X_n}{n} \to \mu \quad \text{as} \quad n \to \infty$$

Let us analyze a simple example. Consider rolling a fair six-sided die with outcomes $\{1, 2, 3, 4, 5, 6\}$, each having an equal probability of $\frac{1}{6}$. Given that we know the outcomes probabilities we can calculate the true expected value: $\mathbb{E}(X) = \sum_{i=1}^{6} x_i P(X = x_i) = \frac{1}{6}(1+2+3+4+5+6) = 3.5$. Now let us suppose we do not have access to the actual probabilities but we are able to draw the die $n = 10$ times, that is, to take 10 samples. Suppose we get the outcomes {3, 5, 2, 6, 4, 1, 5, 6, 3, 4}. The sample mean is $(3 + 5 + 2 + 6 + 4 + 1 + 5 + 6 + 3 + 4)/10 = 3.9$, which is roughly the true expected value of 3.5. As we increase n, the sample mean will approach the true expected value of 3.5.

Properties

Expectation is a linear operator and, thus, satisfies the properties of linearity. Concretely, if X and Y are random variables with finite expected values and a is a constant, then

$$\mathbb{E}[X+Y] = \mathbb{E}[X] + \mathbb{E}[Y],$$
$$\mathbb{E}[aX] = a\mathbb{E}[X],$$

The linearity property extends to multiple random variables $X_1, X_2, \ldots, X_n$:

$$\mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i].$$

The variance of a random variable measures the spread or dispersion of its possible values around the mean (expected value). Specifically, it measures the squared expected deviation from the mean:

$$\mathrm{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - \mathbb{E}[X]^2.$$

1.12 Information Entropy

When analyzing random variables, we are often interested in quantifying uncertainty, that is, how unpredictable the outcomes of the variable are. This uncertainty depends on how the probabilities are distributed across all possible outcomes. If the probabilities are evenly spread, uncertainty is high because every outcome is equally likely. Conversely, when the probabilities are concentrated around a few outcomes, uncertainty is lower because the results are more predictable.

Surprise / Information Content

Let us consider a scenario where we measure the daily temperature in a specific city. If the temperature consistently stays around 25°C, the outcomes are predictable, and uncertainty is low. However, if temperatures range widely, from 15°C to 40°C, with varying probabilities, uncertainty is higher because the potential outcomes are less predictable. To quantify this uncertainty, we need to understand *surprise* and *information content*.

Surprise reflects how unexpected an observed outcome is. For example, if temperatures in the city typically range between 20°C and 30°C, observing a value like 25°C is unsurprising because it aligns with our expectations. On the other hand, observing a rare spike of 40°C is highly surprising. The more surprising an outcome, the more *information content* that outcome conveys regarding the underlying probability distribution. For instance, repeatedly observing 25°C adds little new information because it merely confirms our expectations, whereas observing 40°C significantly updates our understanding of the variability in temperatures.

The idea of less probable events being more surprising and bearing more information content is captured by the following expression (recall that $\ln(a/b) = \ln(a) - \ln(b)$):

$$I(x) \doteq \ln\left(\frac{1}{P(X=x)}\right) = \ln(1) - \ln(P(X=x)) = -\ln(P(X=x)),$$

where x is the observed value taken by random variable X. Intuitively, this expression states that the higher the probability of observing x the lower its information content. The logarithm non-linearly squashes information content as the probability decreases, reflecting the fact that extremely rare events do not carry unreasonable amount of information about the probability distribution. Note that $I(x)$ is not defined for non-occurring events (null probability). If they never occur, how could they have information content?

Example. Suppose the probabilities of two temperature events are $P(T = 25°\mathrm{C}) = 0.8$ (highly probable) and $P(T = 40°\mathrm{C}) = 0.01$ (unlikely). The information content $I(T)$ for an observed temperature $T = t$ is given by $I(t) = -\ln(P(T = t))$. For the highly probable event $T = 25°\mathrm{C}$, the information content is $I(25) = -\ln(0.8) \approx 0.22$, reflecting low surprise. For the unlikely event $T = 40°\mathrm{C}$, the information content is $I(40) = -\ln(0.01) \approx 4.61$, indicating high surprise. This demonstrates that observing rare events provides significantly more information compared to common ones.

Entropy of a Discrete Random Variable

The concept of *entropy* captures the overall uncertainty of a random variable, that is, the average surprise or information content over all possible outcomes. Formally, the entropy of a *discrete* random variable X is defined as:

$$\mathcal{H}(X) = \mathbb{E}[I(x)] = \mathbb{E}[-\ln(P(X = x))] = -\sum_x P(X = x)\ln(P(X = x)). \tag{2}$$

Example. Suppose the discrete random variable T_1 represents the temperature with three possible outcomes: 20°C, 25°C, and 30°C, having probabilities $P(T_1 = 20) = 0.2$, $P(T_1 = 25) = 0.5$, and $P(T_1 = 30) = 0.3$. The entropy $\mathcal{H}(T_1)$ is given by $\mathcal{H}(T_1) = -\sum_t P(T_1 = t)\ln(P(T_1 = t))$. Substituting the probabilities we obtain: $\mathcal{H}(T_1) = -(0.2\ln(0.2)+0.5\ln(0.5)+0.3\ln(0.3)) \approx 1.03$. Now let us apply the same reasoning to a discrete random variable T_2 with much less spread: $P(T_2 = 20) = 0.8$, $P(T_2 = 25) = 0.1$, and $P(T_2 = 30) = 0.1$. In this case, the entropy is $\mathcal{H}(T_2) = -(0.8\ln(0.8)+0.1\ln(0.1)+0.1\ln(0.1)) \approx 0.64$. These examples show that the entropy is higher for spreader probability distributions (1.03 vs. 0.64).

Entropy of a Continuous Random Variable

The summation in Equation 2 can be substituted by an integral to compute the entropy of a *continuous* random variable X with PDF denoted by $f_X(x)$:

$$\mathcal{H}(X) = -\int_{-\infty}^{\infty} f_X(x)\ln(f_X(x))\,dx. \tag{3}$$

The entropy of a Gaussian (normal) distribution X with mean μ and standard deviation σ can be computed by solving the integral from Equation 3 with the Gaussian PDF (Equation 1):

$$\mathcal{H}(X) = \frac{1}{2} + \frac{1}{2}\ln(2\pi) + \ln(\sigma). \tag{4}$$

Note that the entropy of a Gaussian distribution depends on its standard deviation σ. A higher standard deviation (i.e., greater spread) leads to higher entropy, reflecting greater uncertainty in the distribution.

Example. Consider two temperature distributions: one with a mean of 25°C and standard deviation $\sigma_1 = 3$, and another with a mean of 30°C and standard deviation $\sigma_2 = 5$. For the first distribution, substituting $\sigma_1 = 3$ in Equation 4, we get $\mathcal{H}(T_1) = \frac{1}{2} + \frac{1}{2}\ln(2\pi) + \ln(3) \approx 2.52$. For the second distribution, substituting $\sigma_2 = 5$, we get $\mathcal{H}(T_2) = \frac{1}{2} + \frac{1}{2}\ln(2\pi) + \ln(5) \approx 3.03$. The entropy of the second distribution is higher, reflecting greater uncertainty due to its larger standard deviation.

1.13 Kullback-Leibler (KL) Divergence

Kullback-Leibler (KL) divergence measures the difference between the probability distribution P_X of a random variable X and an approximation Q_X, which is also a probability distribution over X. It quantifies how much information is lost when Q_X is used to approximate P_X. A smaller KL divergence indicates that Q_X is more similar to P_X, and thus a better approximation.

1.13.1 Discrete Random Variables

For a discrete random variable X with probability distributions P_X and Q_X, the KL divergence is defined as:

$$D_{KL}(P_X, Q_X) = \sum_x P_X(X = x)\left(\ln(P_X(X = x)) - \ln(Q_X(X = x))\right),$$

where $P_X(X = x)$ and $Q_X(X = x)$ represent the probabilities of X taking the value x under probability distributions P_X and Q_X, respectively. The term $\ln(P_X(X = x)) - \ln(Q_X(X = x))$ represents the difference in information content for the outcome x under the two distributions. Hence, the KL divergence is the *expected value of this difference*, weighted by $P_X(X = x)$, which represents the actual proportion of realizations of x under the true distribution. Recalling that $\ln(a/b) = \ln(a) - \ln(b)$, the KL divergence can be rewritten in its most commonly used formulation:

$$D_{KL}(P_X, Q_X) = \mathbb{E}_{x\sim P_X}\left[\ln\left(\frac{P_X(X=x)}{Q_X(X=x)}\right)\right] = \sum_x P_X(X=x)\ln\left(\frac{P_X(X=x)}{Q_X(X=x)}\right). \tag{5}$$

Example. Consider a discrete random variable X representing weather conditions with the outcomes {Sunny, Cloudy, Rainy}. The true distribution P_X is given as $P_X(X = \text{Sunny}) = 0.7$, $P_X(X = \text{Cloudy}) = 0.2$, $P_X(X = \text{Rainy}) = 0.1$. Two approximate distributions are considered: $Q_{X,1}$ with $Q_{X,1}(X = \text{Sunny}) = 0.5$, $Q_{X,1}(X = \text{Cloudy}) = 0.3$, $Q_{X,1}(X = \text{Rainy}) = 0.2$, and $Q_{X,2}$ with $Q_{X,2}(X = \text{Sunny}) = 0.4$, $Q_{X,2}(X = \text{Cloudy}) = 0.4$, $Q_{X,2}(X = \text{Rainy}) = 0.2$. Using the KL divergence formula, we compute for $Q_{X,1}$: $0.7\ln\left(\frac{0.7}{0.5}\right) \approx 0.2355$, $0.2\ln\left(\frac{0.2}{0.3}\right) \approx -0.0810$, $0.1\ln\left(\frac{0.1}{0.2}\right) \approx -0.0693$. Summing these, we get: $D_{KL}(P_X\|Q_{X,1}) \approx 0.2355 - 0.0810 - 0.0693 = 0.0852$. For $Q_{X,2}$: $0.7\ln\left(\frac{0.7}{0.4}\right) \approx 0.3917$, $0.2\ln\left(\frac{0.2}{0.4}\right) \approx -0.1386$, $0.1\ln\left(\frac{0.1}{0.2}\right) \approx -0.0693$. Summing these, we get: $D_{KL}(P_X\|Q_{X,2}) \approx 0.3917 - 0.1386 - 0.0693 = 0.1838$. The comparison shows that $Q_{X,2}$, with higher divergence, is a poorer approximation of P_X than $Q_{X,1}$.

1.13.2 Continuous Random Variables

To handle a continuous random variable X, the KL divergence is computed by replacing the summation with an integral and using $p_X(X = x)$ and $q_X(X = x)$ as PDFs of P_X and Q_X, respectively:

$$D_{KL}(P_X, Q_X) = \int_{-\infty}^{\infty} p_X(X=x)\ln\left(\frac{p_X(X=x)}{q_X(X=x)}\right)dx. \tag{6}$$

If the probability distributions P_X and Q_X are Gaussian with means (μ_P, μ_Q) and standard deviations (σ_P, σ_Q), the KL divergence is derived by solving the integral from Equation 6 with the Gaussian PDF (Equation 1), resulting in the following expression:

$$D_{KL}(P_X, Q_X) = \ln\left(\frac{\sigma_Q}{\sigma_P}\right) + \frac{\sigma_P^2 + (\mu_P - \mu_Q)^2}{2\sigma_Q^2} - \frac{1}{2}. \tag{7}$$

This formula captures contributions from both the difference in means (μ_P, μ_Q) and standard deviations (σ_P, σ_Q). If the distributions have identical means and variances, the KL divergence is zero, reflecting no difference between the two distributions.

Example. Consider a continuous random variable X following the true Gaussian distribution P_X with $\mu_P = 0$ and $\sigma_P = 1$. Two approximate Gaussian distributions are considered: $Q_{X,1}$ with $\mu_{Q_1} = 1$ and $\sigma_{Q_1} = 1.8$, and $Q_{X,2}$ with $\mu_{Q_2} = 0.5$ and $\sigma_{Q_2} = 1.3$. For $Q_{X,1}$: $D_{KL}(P_X, Q_{X,1}) = \ln\left(\frac{1.8}{1}\right) + \frac{1+(0-1)^2}{2\cdot 1.8^2} - \frac{1}{2} \approx 0.3964$. For $Q_{X,2}$: $D_{KL}(P_X, Q_{X,2}) = \ln\left(\frac{1.3}{1}\right) + \frac{1+(0-0.5)^2}{2\cdot 1.3^2} - \frac{1}{2} \approx 0.1322$. Comparing the divergences, $Q_{X,2}$, with a smaller KL divergence (0.1322 vs 0.3964), is a closer approximation to P_X than $Q_{X,1}$.

1.14 Chain Rule of Calculus

The derivative of the composition of two functions can be found using the chain rule of calculus. This reasoning extends directly to partial derivatives and gradients since gradients are vectors composed of partial derivatives.

Given a function $y = f(u)$ and another function $u = g(x)$, that is, $y = f(g(x))$, the derivative of y with respect to x according to the chain rule is:

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}.$$

The chain rule can be applied recursively. For instance, in the case of three composite functions $y = f(u), u = g(v), v = h(x)$, that is, $y = f(g(h(x)))$, the chain rule applies as follows:

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dv}\frac{dv}{dx}.$$

Let us analyze the example of a quadratic function of a single variable x:

$$y = (3 + 4x)^2.$$

This function can be decomposed as the composite of the following two functions:

$$y = f(u) = u^2$$
$$u = g(x) = 3 + 4x.$$

The derivatives of these functions are:

$$\frac{dy}{du} = f'(u) = 2u = 2(3 + 4x) = 6 + 8x$$
$$\frac{du}{dx} = g'(x) = 4.$$

We can now directly apply the chain rule in order to obtain the derivative of y with respect to x:

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx} = (6 + 8x) \cdot 4 = 24 + 32x.$$

Let us analyze another example, in this case a quadratic function of two variables x and z:

$$y = (3 + 4x + 2z)^2.$$

As before, this function can be decomposed as the composite of the following two functions:

$$y = f(u) = u^2$$
$$u = g(x, z) = 3 + 4x + 2z$$

The next step is to compute the derivatives of these functions. However, as g has two variables we instead compute its *partial derivatives* with respect to each one:

$$\frac{dy}{du} = f'(u) = 2u = 2(3 + 4x + 2z) = 6 + 8x + 4z$$
$$\frac{du}{dx} = g'_x(x, z) = \frac{\partial g}{\partial x}(x, z) = 4$$
$$\frac{du}{dz} = g'_z(x, z) = \frac{\partial g}{\partial z}(x, z) = 2.$$

Finally, by applying the chain rule we are able to obtain the partial derivatives of y with respect to x and z:

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx} = (6 + 8x + 4z) \cdot 4 = 32x + 16z + 24.$$
$$\frac{dy}{dz} = \frac{dy}{du} \cdot \frac{du}{dz} = (6 + 8x + 4z) \cdot 2 = 16x + 8z + 12.$$

2 Markov Decision Processes (MDPs)

Markov Decision Processes (MDPs) are a fundamental framework for modeling sequential decision-making, which serve as an idealized mathematical model for reinforcement learning problems. In an MDP, the decision-making entity is called the agent, while everything external to it, with which it interacts, is known as the environment. The formalism presented in this section closely follows [8].

2.1 Definition

The agent and the environment engage in a continuous cycle. Concretely, at each discrete time step $t = 0, 1, 2, 3, \ldots$, the agent observes the environment's state, S_t, and based on this observation, selects an available action in the current state, A_t. After taking action A_t, at the next time step, the agent receives a numerical reward, R_{t+1}, and the environment transitions to a new state, S_{t+1}. This state transition can occur as a consequence of the agent's action or due to environment's intrinsic dynamics. Figure 2 illustrates the agent-environment interaction cycle in a MDP.

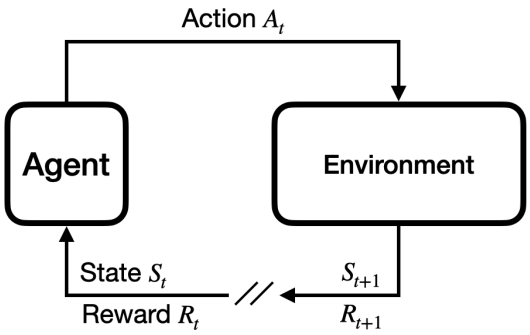


Figure 2: The agent-environment interaction cycle in a MDP [adapted from (Sutton & Barto, 2018)].

The agent-environment iterative process generates an *history* or *trajectory* τ:

$$\tau \doteq (S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots).$$

Actions can be low-level, such as the torques to be applied to the agent's actuator, but they can also be high-level, such as task-level decisions. They can also be about internal cognitive mechanisms, such as controlling the agent's attention. State information can also span from low-level representations, such as pixel RGB values, up to high-level symbolic representations of the environment. State can also include agent specific information, as its position in the environment, and can also include historical information.

MDP assume state S_t, action A_t, and reward R_t to be random variables that take values from the sets $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R} \subset \mathbb{R}$, respectively. These sets encompass all possible states, actions, and reward values. The use of random variables reflect the stochastic nature of the environment and agent-environment interactions.

All states in MDPs exhibit the Markov property, which asserts that the conditional probability distribution of the future state in the stochastic process depends solely on the present state, making the past and future states conditionally independent given the present state. In other words, the current state encapsulates all information from the history that is relevant to determine the probabilities of the future state. Formally, for a given discrete set of random variables $\{X_t\}_{t \in \mathbb{N}_0}$, this is expressed as

$$P(X_{t+1} = x_{t+1} \mid X_t = x_t, X_{t-1} = x_{t-1}, \ldots, X_0 = x_0) = P(X_{t+1} = x_{t+1} \mid X_t = x_t).$$

Complying with the Markov property, it is possible to define the MDP's deterministic *dynamics function* $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ as the probability of the current state S_t and current reward R_t taking the specific values s' and r passed as arguments of the function, given the specific values of the preceding state s and action a also passed as arguments of the function:

$$p(s', r \mid s, a) \doteq P(S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a).$$

Note that p specifies a probability distribution for each choice of s and a and, so:

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) = 1, \quad \text{for all } s \in \mathcal{S}, a \in \mathcal{A}.$$

By marginalising over all reward values in the dynamics function, it is possible to define a *state-transition function* $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ as the probability of reaching state s' by executing action a in state s. This function allows the agent to predict the consequences of its actions. Formally,

$$p(s' \mid s, a) \doteq P(S_t = s' \mid S_{t-1} = s, A_{t-1} = a) = \sum_{r \in \mathcal{R}} p(s', r \mid s, a).$$

We can also define a function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ as the expected reward resulting from choosing action a in a given state s (the probabilities of specific reward values are obtained by marginalising over all possible states in the dynamics function):

$$r(s, a) \doteq \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r \mid s, a).$$

2.2 Example

Consider an MDP where an agent can be in one of two states $S = \{A, B\}$ and can take two actions $\mathcal{A} = \{\text{move}, \text{stay}\}$. State A corresponds to a location that is easy to move away from (e.g., hard terrain), whereas state B corresponds to a location that is very hard to move away from (e.g., sandy terrain). A visual representation of this MDP is provided in Figure 3, while Table 4 presents the corresponding dynamics and reward functions.

If the agent is in state A and selects the move action, then it is highly likely to reach B with probability $P(S_t = B \mid S_{t-1} = A, A_{t-1} = \text{move}) = 0.8$. There is a small chance of slipping and staying in A with probability $P(S_t = A \mid S_{t-1} = A, A_{t-1} = \text{move}) = 0.2$. If the agent opts to stay in state A, it is very likely to remain there with probability $P(S_t = A \mid S_{t-1} = A, A_{t-1} = \text{stay}) = 0.9$. There is a small chance that it will slip to state B with probability $P(S_t = B \mid S_{t-1} = A, A_{t-1} = \text{stay}) = 0.1$.

If the agent is at state B and selects the move action towards state A, it is likely to be able to reach A with probability $P(S_t = A \mid S_{t-1} = B, A_{t-1} = \text{move}) = 0.6$, but there is a significant probability of remaining in B with probability $P(S_t = B \mid S_{t-1} = B, A_{t-1} = \text{move}) = 0.4$. The difficulty of leaving B is reflected in the deterministic consequence of staying in B with probability $P(S_t = B \mid S_{t-1} = B, A_{t-1} = \text{stay}) = 1.0$.

The reward is defined to be 0 everywhere except when the agent decides to stay in state B, where it can consume food and earn a +1 reward. To accumulate reward, the agent needs to learn that when in A, it should move to B and, once reached B, it should stay there forever.

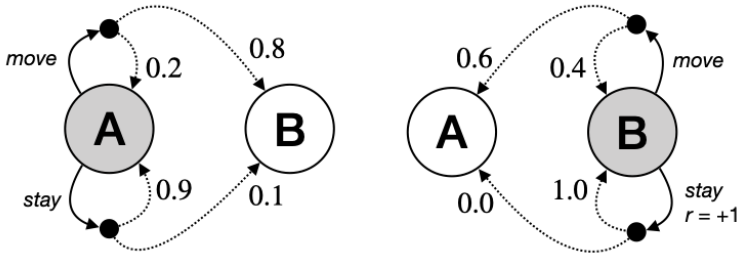


Figure 3: Illustrative MDP example. For simplicity, the visualization of the MDP is split into two parts: the left side depicts the transitions and actions when the agent is in state A, $S_t = A$, and the right side depicts the transitions and actions when the agent is in state B, $S_t = B$. Solid arrows represent actions, while dashed arrows indicate state transitions along with their respective conditional probabilities. The small dots highlight the stochastic outcomes of the actions.

2.3 Rewards and Returns

RL builds upon the idea that it is possible to formulate goals in terms of reward signals, given that the agent then seeks to maximize the amount of reward it receives in the long term. This means that instead

s	a	s'	$p(s' \mid s, a)$	$r(s, a)$
A	move	B	0.8	0
A	move	A	0.2	0
A	stay	A	0.9	0
A	stay	B	0.1	0
B	move	A	0.6	0
B	move	B	0.4	0
B	stay	B	1.0	+1

Table 4: Dynamics and reward functions for the MDP example. Here, $S_t = s'$, $S_{t-1} = s$, and $A_{t-1} = a$.

of explicitly specifying goals in the agent, the designer defines reward functions that, when maximized, induce the agent to accomplish the designer goals. This idea that goals can be defined in terms of reward signals is known as the *reward hypothesis*.

Rewards are scalar numbers that can take negative and positive values. For instance, an agent learning to move on a maze may receive a reward of -1 for every time step so as to induce finding the shortest path. Rewards can also be sparser. For instance, an agent may receive a +10 reward every time it finds a given object, a -10 reward every time it collides against an obstacle, and a 0 in all other situations.

We often define the goal of the agent as to maximize the expected *discounted return*:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k, \tag{8}$$

where T is the final time step and $0 \leq \gamma \leq 1$ is the *discount factor*. Tasks are called *continuing* when $T = \infty$ and *episodic* when otherwise.

If $\gamma = 0$ then the agent only takes into account the immediate reward, meaning that it is not able to take into account that one's action influence also future rewards (e.g., if the agent opts for staying still it will never be able to reach the object that will provide reward). As γ approaches 1 future rewards are progressively being more accounted for, although the more recent the reward is, the more important it is. On the other extreme, if $\gamma = 1$, then all rewards into the future are taken into account, meaning that if $T = \infty$ the return would be infinite. Hence, if $T = \infty$, we need to ensure that $\gamma < 1$.

As mentioned, when $\gamma < 1$, the return is finite, despite the fact it is the sum of infinite terms. For instance, if the agent collects at each time step a constant reward R (note the absence of time-based subscript), the return is a geometric series with the following closed form solution:

$$G_t = \sum_{k=0}^{\infty} R\gamma^k = \frac{R}{1-\gamma} \quad \text{(e.g., if } \gamma = 0.9 \text{ and } R = 1 \text{ then } G_t = 10\text{)}. \tag{9}$$

3 Exact Methods for MDPs

This section explores the problem of solving MDPs using exact methods, which are guaranteed to find the optimal mapping from states to action probabilities. These methods assume complete knowledge of the MDP, which can limit their practical application in real-world problems. However, they form the foundational concepts of reinforcement learning and are crucial for understanding the building blocks of optimal sequential decision making. The formalism presented in this section closely follows [8].

3.1 Policies and Value Functions

The optimal mapping is the one that maximizes the cumulated reward in the long term if strictly followed by the agent, that is, if the agent's actions are fully determined by the optimal mapping. We call *policy* to any of these state-action mappings, even those that are sub-optimal.

Formally, if the agent is in state $s \in \mathcal{S}$ at time t (i.e., $S_t = s$) and is following a *stochastic* policy π, then $\pi(a|s)$ is the probability of the agent selecting action $a \in \mathcal{A}$ at time t (i.e., $A_t = a$). Concretely, $\pi : \mathcal{A} \times \mathcal{S} \to [0, 1]$ is an ordinary function that defines a probability distribution over $a \in \mathcal{A}$ for each $s \in \mathcal{S}$:

$$\sum_{a \in \mathcal{A}} \pi(a|s) = 1 \quad \text{for all } s \in \mathcal{S}.$$

Let us now derive a recursive formulation of the expected *discounted return* so as to better match the sequential nature of decision making, assuming $G_T = 0$:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\ &= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots \right) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

We can use this recursive formulation to derive the *state-value function* of policy π, $v_\pi : \mathcal{S} \to \mathbb{R}$, as the expected discounted return obtained by the agent starting at state $s \in \mathcal{S}$ and following policy π thereafter. The function captures the value or goodness of state s under policy π, thus indicating which states are worth visiting. Formally, v_π is defined as follows:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \left[r + \gamma \mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s'] \right] \\ &= \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \left[r + \gamma v_\pi(s') \right], \quad \text{for all } s \in S, \end{aligned} \tag{10}$$

where $\mathbb{E}_\pi[\cdot]$ is the notation for the expected value of a random variable under the assumption that the agent follows policy π.

Equation 10 is the *Bellman equation* for v_π. In this equation, and according to the chain rule of probabilities, $\pi(a|s)p(s', r|s, a)$ is the joint probability of the triplet a, s', and r, given s. This probability is multiplied by $[r + \gamma v_\pi(s')]$ for every a, s', and r to obtain the expected value (mean) of $[r + \gamma v_\pi(s')]$. Hence, the Bellman equation states that the expected value of the start state must equal the reward obtained at that state plus the (discounted) value of the expected next state (recursive call of v_π). As it will be shown, the Bellman equation is a foundational for computing, estimating, and learning the state-value function v_π.

The Bellman equation form a system of equations, with one equation for each state. For n states, the system consists of n equations with n unknowns. Hence, if we are provided with the dynamics function p, this system can theoretically be solved for v_π. Alternatively, we can solve it using *dynamic programming* techniques, that is, iteratively.

A well-known dynamic programming algorithm for determining v_π is *iterative policy evaluation*. This algorithm starts with an arbitrary initial approximation of v_π, v_0. Then, it sequentially obtains successive approximations based on the immediately preceding ones by using the Bellman equation 10 as an update

rule. Formally, for all $k = 0, 1, 2, \ldots$, approximation v_{k+1} is obtained from the preceding approximation v_k as follows:

$$\begin{aligned} v_{k+1}(s) &\doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \\ &= \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \left[r + \gamma v_k(s')\right], \quad \text{for all } s \in S. \end{aligned} \tag{11}$$

The appeal of *iterative policy evaluation* lies on the guarantee that, if there is a v_π, the sequence $\{v_k\}$ converges to v_π as $k \to \infty$. In practice we stop the sequence as soon as the difference between successive approximations is below a small enough threshold ψ. Algorithm 1 presents the pseudo-code of *iterative policy evaluation*.

Algorithm 1 Iterative Policy Evaluation [adapted from (Sutton & Barto, 2018)]

Input: the policy to be evaluated, π, and the dynamics function, p
Parameters: threshold $\psi > 0$
Initialize arbitrarily $V(s)$, for all $s \in \mathcal{S}^+$, except for $V(terminal) = 0$
Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$**:**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
Until $\Delta < \psi$ // condition occurs when $V(s) \approx v_\pi(s)$ for all $s \in \mathcal{S}$
Return V

We can also define the *action-value function* of policy π, $q_\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ as the expected discounted return obtained by the agent starting at state $s \in \mathcal{S}$, taking action $a \in \mathcal{A}$, and following policy π thereafter:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]. \tag{12}$$

The *state-value function* can be defined as the expected *action-value function* under the action probability distribution induced by the policy:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) q_\pi(s, a). \tag{13}$$

Figure 4 and Figure 5 depict the *backup diagrams* for v_π and q_π, respectively. These diagrams represent the relationships between states, actions, and rewards according to the dynamics and policy functions. Their name stems from the transfer of value information *back* to state / state-action from its successor state / state-action via recursion.

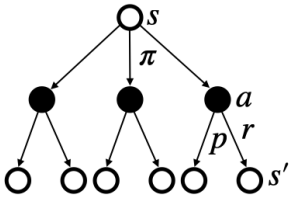


Figure 4: Backup diagram for v_π [adapted from (Sutton & Barto, 2018)].

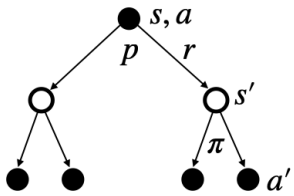
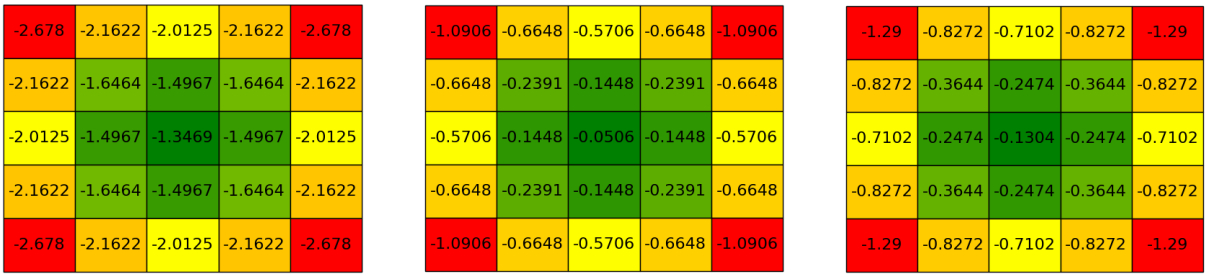


Figure 5: Backup diagram for q_π [adapted from (Sutton & Barto, 2018)].

Example: Gridworld

Let us assume a regular 5×5 grid world in which an agent inhabits. The moves deterministically (i.e., given a state and an action, only one successive state has non-null probability) from its current cell to another by executing an action: north, south, east, or west. Actions that push the agent off the grid



(a) $k = 74$ (at convergence) (b) $k = 3$ (c) $k = 4$

Figure 6: Iterative policy evaluation for grid world with $P(A_t = a) = 0.25, \forall t, a \in \mathcal{A}$.

(a) $k = 78$ (at convergence) (b) $k = 3$ (c) $k = 4$

Figure 7: Iterative policy evaluation with $P(A_t = \text{north}) = 0.25, P(A_t = \text{south}) = 0.25, P(A_t = \text{left}) = 0.15, P(A_t = \text{right}) = 0.35, \forall t$.

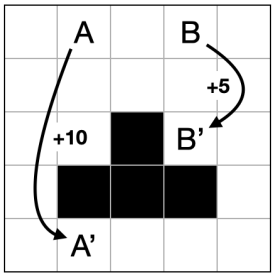
result in no position change and award a reward of -1. The reward is elsewhere 0. The state is the agent's position in the grid world (i.e., cell index). Hence, in this problem, $\mathcal{A} = \{\text{north}, \text{south}, \text{east}, \text{west}\}$ and $\mathcal{S} = \{1, \ldots, 25\}$. Let us assume a discount factor of $\gamma = 0.9$.

Figure 6 shows the grid world with the values of $v_{74}(s), \forall s \in \mathcal{S}$ (left), $v_3(s), \forall s \in \mathcal{S}$ (middle), $v_4(s), \forall s \in \mathcal{S}$ (right), that is, the values of the iterative policy evaluation in its final iteration (convergence with $\psi = 10^{-4}$), in its third iteration and in its fourth iteration. These values were obtained with a uniformly random policy, meaning that all actions are equally probable, $P(A_t = a) = 0.25, \forall t, a \in \mathcal{A}$. The final iteration shows that the further the agent is from the borders, the higher the value of those states, that is, the less likely it is to find the negative rewarding border. Note that $k = 4$ is determined by $k = 3$. For example, the central cell in $k = 4$ is calculated with a weighted average of the north, south, east, and west neighbors in $k = 3$, the weights being the probability of reaching those states times the discount factor: $-0.1448 \times 0.25 \times 0.9 + -0.1448 \times 0.25 \times 0.9 + -0.1448 \times 0.25 \times 0.9 + -0.1448 \times 0.25 \times 0.9 \approx -0.13$.

Figure 7 shows the same procedure but for a different action distribution. In this case the random walk is biased towards moving to right as a consequence of wind presence, $P(A_t = \text{north}) = 0.25, P(A_t = \text{south}) = 0.25, P(A_t = \text{left}) = 0.15, P(A_t = \text{right}) = 0.35, \forall t$. Note how the states with higher value shifted to the left. That is the because starting away from the right boundary renders less likely and delays the encounter with the negative rewards than starting away from the left boundary. Delaying the encounter with rewards impacts the final return due to the discount factor. For the sake of completeness, let us calculate the value of the central cell in $k = 4$, which is done by taking into account the values at iteration $k = 3$: $-0.0665 \times 0.15 \times 0.9 + -0.2695 \times 0.35 \times 0.9 + -0.157 \times 0.25 \times 0.9 + -0.157 \times 0.25 \times 0.9 \approx -0.16$.

Figure 8 shows where iterative policy evaluation converges for a somewhat more complex environment under the uniform random policy, $P(A_t = a) = 0.25, \forall t, a \in \mathcal{A}$. The environment now contains two special cells A and B. Whatever the action taken by the agent in A takes the agent immediately to cell A', awarding a $+10$ reward. Whatever the action taken by the agent in B takes the agent immediately to cell B', awarding a $+5$ reward. Some cells are labelled as walls. Moving to a wall results exactly in the same consequences of moving towards a world boundary: stays in the same state and gets a -1 reward. In the environment without walls, state A, although considered optimal, has an expected return lower than 10, its immediate reward, because the agent transitions from A to A', where it is likely to encounter the boundary of the grid, which awards a negative reward. Conversely, State B is valued higher than 5, its immediate reward, as the agent moves from B to B', which has a positive value. The potential penalty for hitting an edge from B' is outweighed by the potential benefits of reaching A or B from

there.



(a) World rules.

3.3098	8.7901	4.4284	5.3232	1.493
1.5224	2.9931	2.2509	1.9084	0.5482
0.0516	0.739	0.6739	0.359	-0.4024
-0.9728	-0.4347	-0.3541	-0.5848	-1.1823
-1.8569	-1.3444	-1.2285	-1.4221	-1.9744

(b) $k = 47$ (at convergence)

1.6253	6.0516	2.8389	3.7318	0.3865
0.1431	1.4362	1.1057	0.6443	-0.5653
-1.4579	-0.9178		-1.4092	-1.867
-3.1363				-3.3458
-3.9867	-4.3872	-4.5157	-4.4293	-4.0896

(c) $k = 76$ (at convergence)

Figure 8: Iterative policy evaluation for grid world with $P(A_t = a) = 0.25, \forall t, a \in \mathcal{A}$. Left: the rules of the world: jumps positions $A \leftarrow A'$ and $B \leftarrow B'$ and walls positions (black cells). Middle: converged state values considering jumps A and B but without walls. Right: converged state values considering jumps A and B and walls.

For the interested reader, it is important to note that this example is an extension of the classical example presented at (Sutton & Barto, 2018), which only considered uniform random policies, the negative reward in the boundaries, and the positive rewards in the jump states.

3.2 Optimal Policies and Optimal Value Functions

As mentioned, one generally solves RL problems by seeking a policy that maximizes the cumulative reward over time. For finite MDPs, value functions establish a partial ordering among different policies. A policy π is considered superior to or on par with another policy π' if its expected return is at least as high as that of π' for every state. There always exists at least one policy that outperforms or matches all other policies, known as the optimal policy, which we denote by π_*, and whose Bellman equation for its $v_* : \mathcal{S} \to \mathbb{R}$, also known as *Bellman optimality equation*, is

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}} q_{\pi_*}(s, a) \\ &= \max_{a \in \mathcal{A}} \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\ &= \max_{a \in \mathcal{A}} \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\ &= \max_{a \in \mathcal{A}} \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \left[r + \gamma v_*(s') \right]. \end{aligned} \tag{14}$$

Intuitively, the Bellman optimality equation states that the value of a state, under an optimal policy π_*, must correspond to the expected return from executing the best possible action from that state. The term "max" in $\max_{a \in \mathcal{A}} f(a)$ refers to the maximum value of function $f(a)$ when evaluated over all $a \in \mathcal{A}$. Note that the last derivation step of the Bellman optimality equation (Equation 14) does not include the summation over actions present in the Bellman equation (Equation 10). The expectation in the Bellman optimality equation is conditioned on the action, making looping over the actions and weighting them for their probabilities redundant (the probability of the conditioning action would be 1).

Similarly, there is also a Bellman optimality equation for q_*, which recursivelly takes the best possible action in succeeding states:

$$\begin{aligned} q_*(s, a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a \right] \\ &= \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \left[r + \gamma \max_{a'} q_*(s', a') \right]. \end{aligned}$$

Figure 9 and Figure 10 depict the *backup diagrams* for v_* and q_*, respectively. These diagrams include arcs at the agent's choice points to represent that, for v_* and q_*, the maximum over that choice is taken, rather than the expected value given some policy.

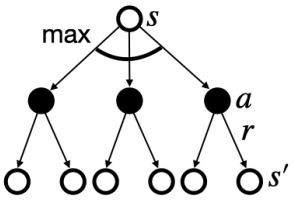


Figure 9: Backup diagram for v_* [adapted from (Sutton & Barto, 2018)].

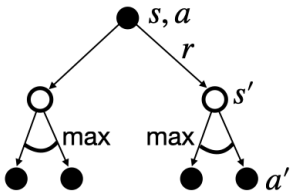


Figure 10: Backup diagram for q_* [adapted from (Sutton & Barto, 2018)].

The Bellman optimality equations form a system of equations, with one equation for each state. For n states, the system consists of n equations with n unknowns. Hence, if we are provided with the dynamics function p, this system can theoretically be solved for v_* and q_*. Note that the Bellman optimality equations make no reference to any specific policy, thus allowing us to build an optimal policy from them, once these have been obtained.

Given that Bellman optimality equations take into account the reward consequences of all possible behavior given a start state s, the optimal action at each instant can be obtained with *greedy* evaluation of v_*, that is, with a one-step search. Hence, this one-step ahead search still encompasses the long-term consequences of the immediate choice. Concretely, the optimal action for state s maximizes the expected sum of the immediate reward and the discounted value of the next state s', summarized in $v_*(s')$:

$$\pi_*(s) = \arg\max_a \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \left[r + \gamma v_*(s') \right]. \tag{15}$$

In this case we are defining our policy as being deterministic, that is, instead of mapping state-actions pairs to probabilities, it maps states to actions: $\pi_* : \mathcal{S} \to \mathcal{A}$. Naturally, provided we have access to $q_*(s, a)$, determining the optimal action for state s becomes even easier. We just have to determine which action a maximizes q_*, which implicitly summarizes the long-term return:

$$\pi_*(s) = \arg\max_a q_*(s, a).$$

As for the *Bellman equation*, we can turn the *Bellman optimality equation* (Equation 14) into an update rule with which we can iteratively approximate the *optimal state-value function* v_*. Formally, for all $k = 0, 1, 2, \ldots$, approximation v_{k+1} is obtained from the preceding approximation v_k as follows:

$$\begin{aligned} v_{k+1}(s) &\doteq \max_{a \in \mathcal{A}} \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \left[r + \gamma v_k(s') \right], \quad \text{for all } s \in S. \end{aligned}$$

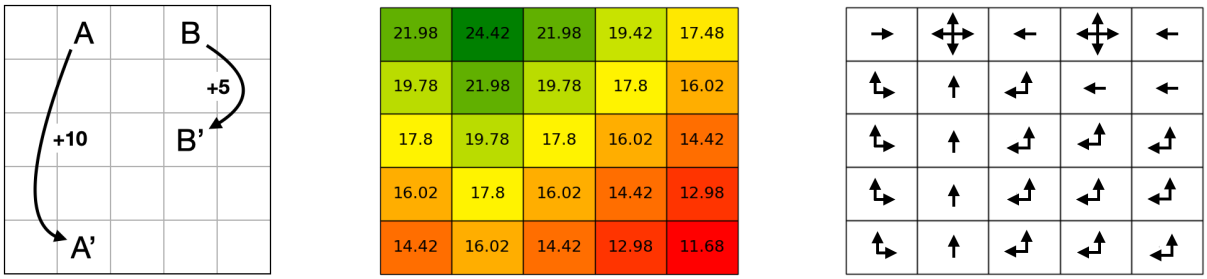
If there is a v_*, this iterative procedure guarantees that the sequence $\{v_k\}$ converges to v_* as $k \to \infty$. As before, we stop the sequence as soon as the difference between successive approximations is below a small enough threshold. Having obtained a sufficiently good approximation of v_*, we can use it to determine the optimal policy π_* by applying Equation 15. This dynamic programming algorithm is known as *value iteration* and its pseudo-code is presented in Algorithm 2

Example: Gridworld (continued)

Let us revisit the grid world example that contains positively rewarding jumps and negatively rewarding boundaries and walls. Figure 11 and Figure 12 show the optimal value function v_* and optimal policy p_* for the grid world without and with walls, respectively. As expected, the optimal actions guide the agent to avoid walls and boundaries while attracting it towards the rewarding jumps in an endless loop. Note that in A and B all actions are optimal, because when reaching those states the agent deterministically jumps to A' and B', respectively, whatever the action it picks.

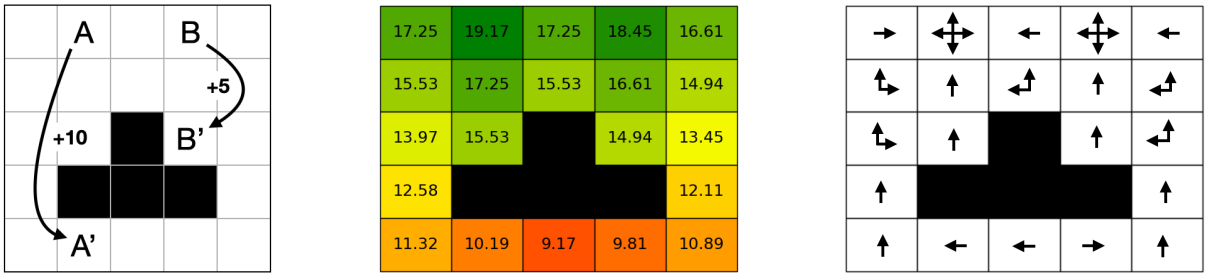
Algorithm 2 Value Iteration [adapted from (Sutton & Barto, 2018)]

Input: the dynamics function, p
Parameters: threshold $\psi > 0$
Initialize arbitrarily $V(s)$, for all $s \in \mathcal{S}^+$, except for $V(terminal) = 0$
// determine V, such that $V \approx v_*$
Loop:
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$**:**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \left[r + \gamma V(s') \right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
Until $\Delta < \psi$
// determine π, such that $\pi \approx \pi_*$
For each $s \in \mathcal{S}$**:**
  $\pi(s) \leftarrow \arg\max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) \left[r + \gamma V(s') \right]$
Return π



(a) World rules. (b) v_* (converged at $k = 111$) (c) π_*

Figure 11: Value iteration for grid world with jumps and no walls. Left: the rules of the world: jumps positions $A \leftarrow A'$ and $B \leftarrow B'$. Middle: optimal value function for all states. Right: optimal policy, where the arrows indicate the optimal actions.



(a) World rules. (b) v_* (converged at $k = 111$) (c) π_*

Figure 12: Value iteration for grid world with jumps and walls. Left: the rules of the world: jumps positions $A \leftarrow A'$ and $B \leftarrow B'$ and walls positions (black cells). Middle: optimal value function for all states. Right: optimal policy, where the arrows indicate the optimal actions.

4 Policy Gradient Methods

4.1 Introduction

As discussed in the previous section, it is theoretically possible to *exactly* solve the system of Bellman optimality equations and, thus, construct optimal policies. Dynamic programming methods, such as *value iteration*, offer practical solutions for computing optimal policies given a perfect finite MDP model of the environment. These approaches have well-defined convergence properties, making them crucial in various domains.

If the agent can observe the full state and there are clearly defined dynamics function p and reward function r, it is logical to solve the system of Bellman optimality equations directly or iteratively via dynamic programming. For instance, this would be the case for a virtual character inhabiting a grid world (or discretized continuous world) with perfectly modeled dynamics and reward functions.

Knowing the consequences of its actions (via the known dynamics function) and which state-action pairs yield rewards (via the known reward function), the agent can solve the Bellman optimality equation via dynamic programming to plan the optimal course of action. This planning procedure occurs in the agent's "brain" before acting in the environment. Being fully knowledgeable, the agent has no need for trial-and-error.

However, what if the agent cannot predict the outcome of its actions or is unaware of which state-action pairs yield higher rewards? This often happens with agents that need to solve complex tasks in hard-to-model environments (e.g., when physics is involved, when there are other agents in the environment, or when the environment is not fully observable). In such cases, the agent lacks sufficient information to solve the Bellman optimality equation via dynamic programming, meaning it cannot plan its course of action in advance. Thus, the agent must try actions in the environment and learn their consequences in terms of state changes and rewards, that is, learn value functions, dynamics functions, and policies through interaction with the environment.

Optimally solving discrete MDPs requires processing all state-action pairs in detail. In realistic discrete problems with large state-action spaces (e.g., chess) or continuous problems (e.g., locomotion control under physics), this results in very large state-action spaces, incurring exponentially large computational requirements, known as the *curse of dimensionality*. Moreover, if a trial-and-error approach is used to learn how to interact with the environment, the agent would need to visit all states, even those that are very similar and close to each other.

To enhance learning and enable sensible decisions in previously unseen states, the agent should be able to *generalize* information gathered from observed similar states to unobserved states. This can be achieved through function approximation, where instead of storing all state information in a table, a parameterized function returns the stored values given the state. Here, we can leverage the generalization capabilities of well-known machine learning function approximation techniques, such as neural networks.

Approximation-based algorithms

Approximation-based algorithms in reinforcement learning can be broadly categorized into two types: *model-free* and *model-based* methods.

Model-free methods do not use a model of the environment. Instead, they learn directly from interactions with the environment. These methods are generally simpler to implement and tune, making them more robust and easier to apply to a wide range of problems. However, they tend to be less sample-efficient, meaning they require the agent to collect more samples in the environment to achieve good performance.

Model-based methods, on the other hand, use a model of the environment to simulate interactions. This allows the algorithm to learn from these simulations, potentially improving sample efficiency. In other words, model-based methods can achieve good performance with fewer samples collected in the environment, which is particularly important in scenarios where collecting data is expensive, time-consuming, or difficult. These models can either be assumed to exist a priori or be learned by the agent as it interacts with the environment. Despite their potential for higher sample efficiency, model-based methods can be more complex to implement and require accurate modeling of the environment. In complex scenarios, it may be harder to learn a model of the environment than to learn the optimal mapping between states and actions.

In reinforcement learning, algorithms can also be categorized into *on-policy* and *off-policy* methods, which are briefly compared next.

On-policy methods use data collected from the current policy to update that same policy. This means that the policy being improved is the same one that is used to generate the data. Examples of on-policy methods include REINFORCE [9] and Proximal Policy Optimization (PPO) [7]. *Off-policy* methods, on the other hand, use data collected from a different policy (behavior policy) to update the policy being optimized (target policy). Examples of off-policy methods include Deep Q-Networks (DQN) [4], Deep Deterministic Policy Gradient (DDPG) [3], and Soft Actor-Critic (SAC) [2].

On-policy methods tend to be more stable because they directly optimize the policy that is being used to interact with the environment. These methods are often simpler to implement since they do not require maintaining separate behavior and target policies. On-policy methods learn directly from the current policy's experiences, which can be beneficial in environments where the policy needs to adapt quickly to changes. When sample collection is inexpensive and efficient, on-policy algorithms can indeed be faster in terms of wall clock time. This is because they continuously update the policy with fresh data from the current policy, leading to quicker convergence.

Off-policy methods are generally more sample-efficient because they can reuse past experiences multiple times to improve the policy. This is particularly useful in environments where data collection is expensive or time-consuming. These methods can explore more effectively by using a behavior policy that encourages exploration while optimizing a target policy that focuses on exploitation. Off-policy methods can learn from a variety of sources, including demonstrations or data generated by other policies, making them versatile in different scenarios.

Outline

Given that *model-free*, *on-policy* methods are typically more stable, simpler to implement, and run in shorter wall clock times, they are ideal for pedagogical purposes. In addition, these methods have been pivotal in many of the latest breakthroughs in the field of deep reinforcement learning, making their learning essential for those who wish to contribute to the field. For these reasons, In this section, we explore *model-free*, *on-policy* methods for learning a *parameterized policy* through trial-and-error, known as *policy gradient methods*. A parameterized policy is a function governed by a set of learnable parameters. The objective of policy gradient methods is to approximate the optimal set of parameters by iteratively improving them as the agent interacts with the environment. We will focus our analysis in the episodic case. The continuing case follows a similar yet more elaborate derivation and analysis. The formalism presented in this section closely follows (Sutton & Barto, 2018).

4.2 Parameterized Policies

We denote the policy parameters as a vector $\boldsymbol{\theta} \in \mathbb{R}^d$, where d is the number of real-valued parameters. The policy is defined as an ordinary function $\pi : \mathcal{A} \times \mathcal{S} \times \mathbb{R}^d \to [0, 1]$, which is defined as the probability that action $a \in \mathcal{A}$ is taken by the agent at time t, given the environment is in state $s \in \mathcal{S}$ and the policy parameters are $\boldsymbol{\theta}$:

$$\pi(a \mid s, \boldsymbol{\theta}) \doteq P(A_t = a \mid S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}).$$

As in the non-parametric case, $\pi(\cdot \mid s, \boldsymbol{\theta})$ defines a probability distribution over $\mathcal{A}$ for each $s \in \mathcal{S}$ given the function's parameterization $\boldsymbol{\theta}$. The agent's action at time t is determined by sampling from this distribution:

$$A_t \sim \pi(\cdot \mid S_t, \boldsymbol{\theta}).$$

Action selection via sampling from the policy-induced action distribution ensures that more rewarding actions are selected more frequently. Occasionally selecting less rewarding actions allows the agent to explore alternatives and gather information about the reward distribution. Balancing the exploration of new actions to gather information about their rewards and exploiting known actions to maximize cumulative reward is a cornerstone of reinforcement learning. Sampling from the policy distribution effectively manages this trade-off.

Policy gradient methods can work with any policy parameterization, provided that $\pi(a \mid s, \boldsymbol{\theta})$ is differentiable with respect to its parameters $\boldsymbol{\theta}$.

4.3 Function Optimization with Gradient Ascent

A recurring task in machine learning algorithms is to iteratively find the parameter vector $\boldsymbol{\theta} \in \mathbb{R}^n$ that locally maximizes a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$, $\arg\max_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$, given an initial guess $\boldsymbol{\theta}_0$. One of the simplest approaches to this problem is known as *gradient ascent*, which makes use of the partial derivatives around the current guess to determine the next best guess, in an iterative fashion.

As we will learn in subsequent sections, *policy gradient* methods build on gradient ascent to update policy parameters. To better understand these methods, this section provides a short machine learning primer using general terms, that is, without any explicit reference to policy gradient algorithm or nomenclature.

The gradient of f at a point $\boldsymbol{\theta} \doteq (\theta_1, \theta_2, \ldots, \theta_n) \in \mathbb{R}^n$ is denoted by $\nabla f(\boldsymbol{\theta})$ and is defined as a vector containing the function's partial derivatives with respect to $\boldsymbol{\theta}$ (recall that a partial derivative is just the derivative with respect to one of the variables while considering the other variables as constants):

$$\nabla f(\boldsymbol{\theta}) \doteq \left(\frac{\partial f}{\partial \theta_1}(\boldsymbol{\theta}), \frac{\partial f}{\partial \theta_2}(\boldsymbol{\theta}), \ldots, \frac{\partial f}{\partial \theta_n}(\boldsymbol{\theta}) \right).$$

Intuitively, the gradient of a function is a vector that represents the direction of steepest ascent of function $f(\boldsymbol{\theta})$. That is, this vector indicates how much f changes as each component θ_i of $\boldsymbol{\theta}$ is varied. The direction of the gradient vector points in the direction of the greatest rate of increase of the function, whereas the magnitude of the vector represents the function's rate of change in that direction.

Starting from an initial guess, $\boldsymbol{\theta}_0$, gradient ascent iteratively takes a small step in the direction of the gradient at each discrete time step $t = 0, 1, 2, 3, \ldots$, as it is the direction in parameter space that induces the fastest incrase in f. The steps are taken with a magnitude that is proportional to the magnitude of the gradient, that is, the steeper the function, the larger the step. Formally, a gradient ascent steps is defined as follows:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \nabla f(\boldsymbol{\theta}_t),$$

where $\alpha \in \mathbb{R}_+$ is a small enough *step size*, also known as *learning rate*. If α is too small, the convergence will be slow. A high α may lead to overshooting and divergence. With a sufficiently small α and ensuring that certain assumptions on the function hold (e.g., continuity) the method is guaranteed to converge to a local maximum. Under stronger assumptions on the function (e.g., convexity), the method can converge to the global maximum.

The iterative process is repeated until some termination criterion is achieved, such as reaching a maximum number of iterations or the change in the function value or the parameter vector is below a certain threshold ϵ:

$$|f(\boldsymbol{\theta}_{t+1}) - f(\boldsymbol{\theta}_t)| < \epsilon \quad \text{or} \quad ||\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t|| < \epsilon.$$

To demonstrate an iteration of gradient ascent, let us assume a function $f(\boldsymbol{\theta}) = 10 - \theta_1^2 - 3\theta_2^2$ with $\boldsymbol{\theta} = (\theta_1, \theta_2)$, plotted in Figure13. The gradient of this function is

$$\nabla f(\boldsymbol{\theta}) = \left(\frac{\partial f}{\partial \theta_1}(\boldsymbol{\theta}), \frac{\partial f}{\partial \theta_2}(\boldsymbol{\theta}) \right) = (-2\theta_1, -6\theta_2).$$

At point $\boldsymbol{\theta}_0 = (1, 1)$, $f(\boldsymbol{\theta}_0) = 10 - 1^2 - 3 \times 1^2 = 6$, and $\nabla f(\boldsymbol{\theta}_0) = (-2 \times 1, -6 \times 1) = (-2, -6)$. Note that the second coordinate of the gradient is three times higher than the first, indicating that a change in parameter θ_2 induces a change in f that is three times higher than a change in parameter θ_1. Hence, if we wish to maximize f, we should move more strongly along θ_2 then along θ_1. We can take such a *gradient ascent* step with $\alpha = 0.05$ as follows: $\boldsymbol{\theta}_1 = \boldsymbol{\theta}_0 + 0.05\nabla f(\mathbf{x_0}) = (1, 1) + 0.05(-2, -6) = (0.9, 0.7)$. We can now evaluate f in this new point, $f(\boldsymbol{\theta}_1) = 10 - 0.9^2 - 3 \times 0.7^2 = 7.72$. Hence, by moving from $\boldsymbol{\theta}_0$ to $\boldsymbol{\theta}_1$ we have climbed f from 6 to 7.72. Across iterations, this procedure slides both θ_1 and θ_2 in the direction of 0, where the function has its maximum, 10. Figure 14 shows the 2D contour plot of the function with the first 35 gradient ascent vectors represented as red arrows. Note how progress in parameter space is faster along $\boldsymbol{\theta}_2$ (explained by the larger partial derivative) and how the algorithm is able to converge to the function's maximum at $\theta_1 = \theta_2 = 0$.

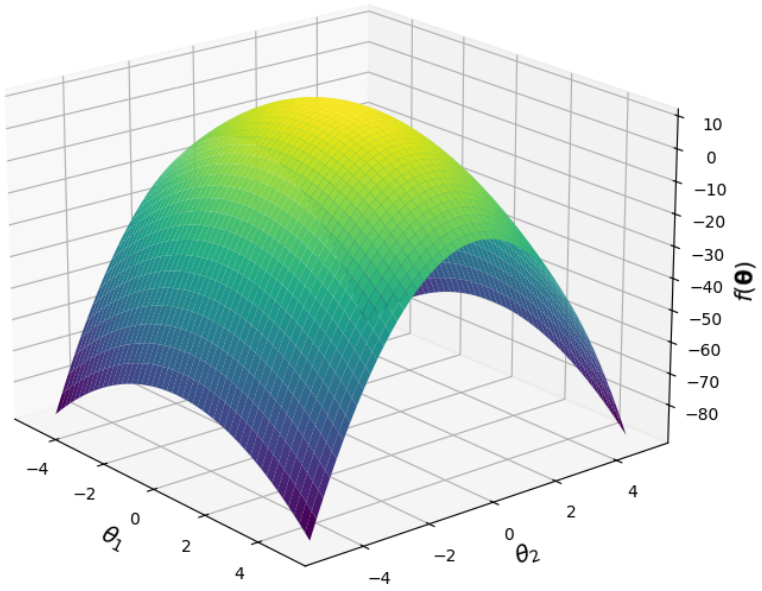


Figure 13: 3D surface plot of the function $f(\boldsymbol{\theta}) = 10 - \theta_1^2 - 3\theta_2^2$.

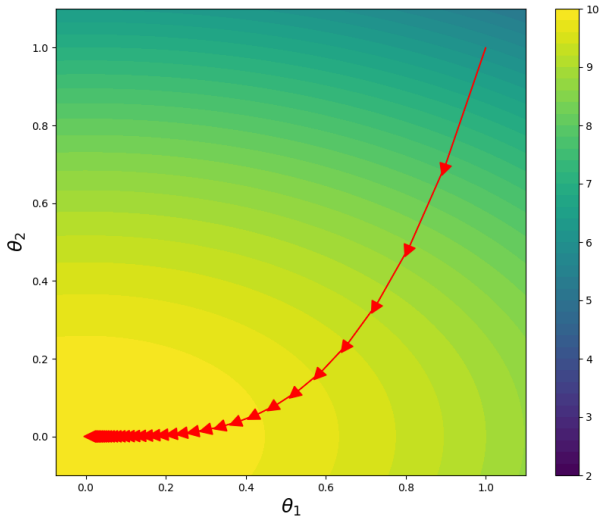


Figure 14: 2D contour plot of function $f(\boldsymbol{\theta}) = 10 - \theta_1^2 - 3\theta_2^2$, with gradient vectors as red arrows.

Gradient Descent

In some cases, instead of maximizing a given function, we may need to minimize it. In the latter case, we use *gradient descent* instead of *gradient ascent*. The iterative procedure is essentially the same, with the exception that the parameters are updated in the opposite direction of the function's gradient:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla f(\boldsymbol{\theta}_t),$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is a differentiable function of the parameters $\boldsymbol{\theta}$, α is the learning rate, and $\boldsymbol{\theta}_0$ is the initial guess for the parameters. The remaining procedures and convergence analysis follow those provided for the gradient ascent case.

Learning Rate Decay

When optimizing a function, it is often beneficial to lower the learning rate as the optimization progresses, that is as t varies. This approach allows for a more extensive exploration of the parameter space in the initial stages and gradually reduces the step sizes to refine the policy.

There are various types of learning rate update schedules. One of the most commonly used is known as *exponential decay*. In this method, the learning rate is adjusted either at every time step (for a smooth decay) or at specified intervals δt (for a staircase decay) before the optimization step. This adjustment is based on an initial learning rate α_0 and an exponential decay factor τ (e.g., updating the learning rate by $\tau = 0.9$ every $\delta t = 100$ time steps):

$$\alpha_t = \alpha_0 \tau^{\frac{t}{\delta t}}. \tag{16}$$

4.4 Policy Gradient

Let us define a policy performance measure $J(\boldsymbol{\theta}) \in \mathbb{R}$, which evaluates how well the policy assigns higher probabilities to actions that result in a high accumulation of reward in the long term, and lower probabilities to actions that result in a low accumulation of reward. The gradient of this measure, $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \in \mathbb{R}^d$, with respect to the policy parameters $\boldsymbol{\theta}$, indicates how changes in the policy parameters influence the performance measure. Therefore, we can gradually maximize performance by adjusting the policy parameters in the direction of the gradient vector of the performance measure.

Performance, and consequently its gradient, can be estimated by running the agent in the environment according to the policy. The obtained stochastic estimate of the gradient, denoted by $\widehat{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})}$, approximates the actual gradient $J(\boldsymbol{\theta})$ in expectation; that is, the more the agent interacts with the environment, the more accurate the estimate becomes. Formally, after a given agent-environment interaction at time t, the policy parameters can be updated by taking a *stochastic gradient ascent* step in $J(\boldsymbol{\theta})$ of size α (the vector $\widehat{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}$ points in the direction in parameter space that induces the fastest increase in $J(\boldsymbol{\theta})$):

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \widehat{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}. \tag{17}$$

4.5 Policy Gradient Theorem

The Policy Gradient Theorem offers an analytic expression for the performance gradient with respect to the policy parameters, which we need to approximate to perform gradient ascent in $J(\boldsymbol{\theta})$ (Equation 17). For the episodic case, the theorem is derived from the following performance definition:

$$J(\boldsymbol{\theta}) \doteq v_{\pi_{\boldsymbol{\theta}}}(s_0) \tag{18}$$

where s_0 is the episodes' initial state and $v_{\pi_{\boldsymbol{\theta}}}$ is the true value function for the policy determined by $\boldsymbol{\theta}$, $\pi_{\boldsymbol{\theta}}$. Maximizing this performance measure means finding the policy whose value function for initial state s_0 is maximal, hence, whose expected return is maximal: there is no other policy that would gather higher reward in the long-term.

Derived from Equation 18 by applying some calculus and by taking into account Equation 13, the *policy gradient theorem* for the episodic case establishes that:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \propto \sum_{s \in \mathcal{S}} \mu_{\pi_{\boldsymbol{\theta}}}(s) \sum_{a \in \mathcal{A}} q_{\pi_{\boldsymbol{\theta}}}(s, a) \nabla_{\boldsymbol{\theta}} \pi(a \mid s, \boldsymbol{\theta}),$$

where $\mu_{\pi_{\boldsymbol{\theta}}}$ is the state visitation probability distribution over $\mathcal{S}$ under $\pi_{\boldsymbol{\theta}}$ (i.e., $\mu_{\pi_{\boldsymbol{\theta}}}(s)$ is the probability of visiting state s) and $\propto$ signifies "proportional to". For the episodic scenario, the constant of proportionality is the average episode length, whereas for the continuing scenario it is 1.

The formal proof of the policy gradient theorem can be found in (Sutton & Barto, 2018). However, we can discuss some informal intuition underlying the theorem. According to Equation 13, the term $\sum_{a\in\mathcal{A}} \pi(a \mid s) q_\pi(s,a)$ represents the value of state s (i.e., the expected discounted return) under policy π, denoted as $v_\pi(s)$. If we disregard the gradient for now, the performance objective $J(\boldsymbol{\theta})$ can be interpreted as a weighted average of the values of all possible states, where the weights are the probabilities of visiting those states under the policy π. This aligns with the intuition that we are optimizing the performance objective to find the policy that induces the highest possible expected return. The gradient of $J(\boldsymbol{\theta})$ is simplified by assuming that only the policy π depends on the parameters $\boldsymbol{\theta}$, while all other elements in the system are treated as constants.

Note that the policy gradient equation contains a sum over all states weighted by their occurrence probability, i.e., it calculates an expected value over states according to the probabilities induced by $\pi_{\boldsymbol{\theta}}$. Given that the agent acts in the environment according to $\pi_{\boldsymbol{\theta}}$, at at each instant t it will sample (experience) the environment state according to those probabilities, $S_t \sim \pi_{\boldsymbol{\theta}}$. Hence, we can simplify the equation by replacing s with the random variable S_t and explicitly expressing the expectation under $\pi_{\boldsymbol{\theta}}$:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &\propto \sum_{s\in\mathcal{S}} \mu_{\pi_{\boldsymbol{\theta}}}(s) \sum_{a\in\mathcal{A}} q_{\pi_{\boldsymbol{\theta}}}(s,a) \nabla_{\boldsymbol{\theta}} \pi(a \mid s, \boldsymbol{\theta}) \\
&= \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[\sum_{a\in\mathcal{A}} q_{\pi_{\boldsymbol{\theta}}}(S_t,a) \nabla_{\boldsymbol{\theta}} \pi(a \mid S_t, \boldsymbol{\theta}) \right] \quad \text{(replacing } s \text{ by the sample } S_t \sim \pi_{\boldsymbol{\theta}}\text{)}.
\end{aligned}$$

The equation can be further simplified. First, we multiply and divide by $\pi(a \mid S_t, \boldsymbol{\theta})$, causing no effect on the equality. This simple trick turns the right-hand side of the equation into an expectation over actions under $\pi_{\boldsymbol{\theta}}$: it includes a sum over actions weighted by their probabilities $\pi(a \mid S_t, \boldsymbol{\theta})$. As for s, we substitute a for sample $A_t \sim \pi_{\boldsymbol{\theta}}$ and explicitly express the expectation under $\pi_{\boldsymbol{\theta}}$. Finally, we include the expected return G_t based on Equation 12. Formally,

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[\sum_{a\in\mathcal{A}} \pi(a \mid S_t, \boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(S_t,a) \frac{\nabla_{\boldsymbol{\theta}} \pi(a \mid S_t, \boldsymbol{\theta})}{\pi(a \mid S_t, \boldsymbol{\theta})} \right] \quad \text{(multiplying and dividing by } \pi_{\boldsymbol{\theta}}\text{)} \\
&= \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[q_{\pi_{\boldsymbol{\theta}}}(S_t,A_t) \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t \mid S_t, \boldsymbol{\theta})}{\pi(A_t \mid S_t, \boldsymbol{\theta})} \right] \quad \text{(replacing } a \text{ by the sample } A_t \sim \pi_{\boldsymbol{\theta}}\text{)} \\
&= \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t \mid S_t, \boldsymbol{\theta})}{\pi(A_t \mid S_t, \boldsymbol{\theta})} \right] \quad \text{(because } q_{\pi_{\boldsymbol{\theta}}}(S_t,A_t) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[G_t \mid S_t, A_t]\text{)}. \qquad (19)
\end{aligned}$$

4.6 The REINFORCE Algorithm

As discussed in Section 1.11, expected values can be approximated by sampling from the underlying distribution and taking the mean of these samples. Hence, we can approximate the expected value of the quantity inside the brackets in Equation 19 and, thus, the gradient $\nabla J(\boldsymbol{\theta})$, by sampling from the underlying distribution $\pi_{\boldsymbol{\theta}}$. Sampling this distribution involves running the agent in the environment according to policy $\pi_{\boldsymbol{\theta}}$ over an episode. The states S_t and actions A_t, for all $t \in \{0, T-1\}$, when plugged into the quantity inside the brackets, generate a set of samples whose mean approaches the true expected value of the quantity inside the brackets.

We are now ready to update Equation 17 with the sampling-based estimated gradient from Equation 19. Concretely, at each time step $t \in \{0, T-1\}$ of every episode of length T, the agent samples the environment by running $\pi_{\boldsymbol{\theta}}$ and updates the policy parameters $\boldsymbol{\theta}$ with learning rate α as follows:

$$\begin{aligned}
\boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t \mid S_t, \boldsymbol{\theta}_t)}{\pi(A_t \mid S_t, \boldsymbol{\theta}_t)} \\
&= \boldsymbol{\theta}_t + \alpha G_t \nabla_{\boldsymbol{\theta}} \ln \pi(A_t \mid S_t, \boldsymbol{\theta}_t) \quad \text{(because } \nabla_{\boldsymbol{\theta}} \ln x = \nabla x / x\text{)}. \qquad (20)
\end{aligned}$$

Let us shortly analyze the intuition behind the update rule (Equation 20). The gradient $\nabla_{\boldsymbol{\theta}} \ln \pi(A_t \mid S_t, \boldsymbol{\theta}_t)$ is a vector defined in policy parameter space that points in the direction of higher variation in

the policy function $\pi(A_t \mid S_t, \boldsymbol{\theta}_t)$ for the given S_t and A_t. That is, summing this vector to the current policy parameters $\boldsymbol{\theta}_t$ will push them in such a way that the policy function $\pi(A_t \mid S_t, \boldsymbol{\theta}_t)$ will return a higher value (probability) for the given S_t and A_t. In other words, it will raise the probability of selecting the action A_t, when visiting state S_t. Given that the gradient is multiplied by the return G_t, when the latter is negative, the effect is the opposite, the parameters $\boldsymbol{\theta}_t$ are pushed in the opposite direction of the gradient $\nabla_{\boldsymbol{\theta}} \ln \pi(A_t \mid S_t, \boldsymbol{\theta}_t)$, in such a way that the policy function $\pi(A_t \mid S_t, \boldsymbol{\theta}_t)$ will return a lower value (probability) for the given S_t and A_t. This way the parameters $\boldsymbol{\theta}_t$ are adjusted in order to raise the probability of selecting actions correlated with high return and to lower the probability of selecting actions correlated with low return.

The *model-free*, *on-policy* algorithm we have been describing is known as REINFORCE and its pseudo-code for the episodic case, augmented with an exponentially decaying learning rate, is presented in Algorithm 3. Note that the equations in the algorithm include discount factor $0 \leq \gamma \leq 1$, which was left out during the algorithm's description for the sake of clarity.

Algorithm 3 Episodic REINFORCE Algorithm with decaying learning rate

Input: a differentiable policy parameterization $\pi(a \mid s, \boldsymbol{\theta})$,
Parameters: base learning rate $\alpha_0 > 0$, discount factor $0 \leq \gamma \leq 1$, episode length T
Parameters: learning decay rate, $0 < \tau \leq 1$, decay interval, $\delta t > 0$

Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^d$ (e.g., randomly)
Initialize global time step, $n \leftarrow 0$
Loop forever:
  // Generating an episode (policy rollout)
  Observe initial state S_0
  For each time step $t = 0, 1, \ldots, T-1$:
    Sample action from policy, $A_t \sim \pi(\cdot \mid S_t, \boldsymbol{\theta})$
    Take action A_t, observe reward R_{t+1} and next state S_{t+1}
  // Policy update based on episode return
  For each step of the episode $t = 0, 1, \ldots, T-1$:
    $n \leftarrow n + 1$
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\alpha \leftarrow \alpha_0 \tau^{\frac{n}{\delta t}}$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t \mid S_t, \boldsymbol{\theta})$

4.7 Optimality and Variance

In the REINFORCE algorithm, the expected update over the course of an episode aligns with the direction of the performance gradient. This guarantees an expected improvement in performance for sufficiently small step size (also known as *learning rate*) α and ensures convergence to a local optimum under standard conditions of stochastic approximation with a decreasing α.

REINFORCE is a *Monte Carlo algorithm* because it estimates returns by sampling complete episodes under the current policy π_θ. This involves generating sequences of states, actions, and rewards through the agent's interactions with the environment according to the current policy π_θ. By averaging these sampled returns over many episodes, REINFORCE approximates the expected return for each action-state pair to inform policy updates.

While REINFORCE approximates the expected return, the algorithm often experiences high variance in its gradient estimates due to the variability of returns across episodes, resulting in slow learning. This high variance arises because the return G_t depends on the entire sequence of actions and rewards, which can differ significantly between episodes. As a result, the learning process can be noisy and may require a large number of episodes to effectively approximate the optimal policy.

4.8 Baselines

A popular approach to reduce variance in the gradient estimate and, thus, to render policy learning faster and more stable, is to generalize the policy gradient theorem to include a comparison of the action value to an arbitrary a baseline function of state $b(s)$:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \propto \sum_{s \in \mathcal{S}} \mu_{\pi_{\boldsymbol{\theta}}}(s) \sum_{a \in \mathcal{A}} \left(q_{\pi_{\boldsymbol{\theta}}}(s,a) - b(s)\right) \nabla_{\boldsymbol{\theta}} \pi(a \mid s, \boldsymbol{\theta}),$$

The baseline can be any function, provided it does not vary with a. By being independent of a, it can be factored out of the summation over actions, which as a result sums to 1 due to the properties of a probability distribution. Since the gradient of a constant is zero, the policy gradient theorem remains valid because the contribution of the baseline term is zero:

$$\sum_{a \in \mathcal{A}} b(s) \nabla_{\boldsymbol{\theta}} \pi(a \mid s, \boldsymbol{\theta}) = b(s) \nabla_{\boldsymbol{\theta}} \sum_{a \in \mathcal{A}} \pi(a \mid s, \boldsymbol{\theta}) = b(s) \nabla_{\boldsymbol{\theta}} 1 = 0$$

Hence, using a baseline function does not alter the expected value of the gradient under the condition that the baseline function is independent of the action taken. This ensures that the baseline only affects the variance of the gradient, reducing it, and not its expectation. The policy gradient theorem with baseline can be used to derive an update rule for the REINFORCE algorithm:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \left(G_t - b(S_t)\right) \nabla_{\boldsymbol{\theta}} \ln \pi(A_t \mid S_t, \boldsymbol{\theta}_t).$$

A natural choice for the baseline function is a differentiable approximation of the state-value function, $\hat{v}(S_t, \mathbf{w})$, parameterized by a vector $\mathbf{w} \in \mathbb{R}^m$:

$$b(S_t) \doteq \hat{v}(S_t, \mathbf{w}).$$

By utilizing $G_t - \hat{v}(S_t, \mathbf{w})$ in the policy gradient update, we reinforce actions that result in higher returns than the average return under the current policy. Specifically, actions leading to $G_t - \hat{v}(S_t, \mathbf{w}) > 0$ increase in probability, making the agent more likely to choose them in the future and thereby improving the policy. When the sample return G_t matches the expected return $\hat{v}(S_t, \mathbf{w})$, their difference becomes zero, resulting in no change to the action probabilities. This mechanism inherently stabilizes the learning process by avoiding unnecessary updates. Directly using the return G_t would lead to high-variance gradient estimates, which may cause instability and slow learning. Subtracting the baseline $\hat{v}(S_t, \mathbf{w})$ reduces the overall magnitude of the updates, leading to lower variance and more stable and efficient learning.

4.9 State-Value Function Approximation

Let us now discuss how the agent can incrementally learn an approximation of the true state-value function, $\hat{v}(s, \mathbf{w}) \approx v_{\pi_{\boldsymbol{\theta}}}(s)$, as it interacts with the environment. To improve this approximation, we first need to define a measure of the difference between the approximate values and the true values, which is known as an error measure. A common error measure is the *mean squared value error*, defined as follows:

$$\overline{VE}(\mathbf{w}) \doteq \sum_{s \in \mathcal{S}} \mu(s) \left(v_{\pi_{\boldsymbol{\theta}}}(s) - \hat{v}(s, \mathbf{w})\right)^2 \tag{21}$$

$$= \mathbb{E}_{\pi_{\boldsymbol{\theta}}} \left[\left(v_{\pi_{\boldsymbol{\theta}}}(S_t) - \hat{v}(S_t, \mathbf{w})\right)^2\right] \quad \text{(replacing } s \text{ by the sample } S_t \sim \pi_{\boldsymbol{\theta}}\text{)} \tag{22}$$

where $\mu(s)$ is the steady-state probability of visiting state s. By weighting the error with state visitation probability, this error measure places greater importance on prediction errors in frequently visited states compared to those in seldomly visited states. Therefore, even if $\hat{v}$ cannot accurately map all states to their correct values due to limited capacity (i.e., a smaller number of parameters compared to the total number of states), it will focus on minimizing errors in the states that are actually visited. The set of states that are frequently visited depends heavily on the policy $\pi_{\boldsymbol{\theta}}$ and may be sparse.

The squared term in the equation ensures that all errors are positive and maintains differentiability. Additionally, it emphasizes larger errors more than smaller ones, directing the optimization process to focus on reducing significant errors in the search space.

The weighted sum of the squared errors in Equation 21 has the signature of an expected value, explicitly stated in Equation 22. As already discussed, expected values can be approximated by sampling from the underlying distribution and taking the mean of these samples. This process involves running the agent in the environment according to policy $\pi_{\boldsymbol{\theta}}$ over an episode. The states S_t and actions A_t, for all $t \in \{0, T-1\}$, when plugged into the quantity inside the brackets, generate a set of samples whose mean approaches the true expected value of the quantity inside the brackets.

We can define an update rule for the state-value approximation function as we did for policy function, that is, that uses the gradient of the function we are optimizing. Given that in this case the function is a cost, also known as a *loss function*, the problem is one of minimization. Thus, we approach it with *stochastic gradient descent*:

$$\begin{aligned}\mathbf{w}_{t+1} &\doteq \mathbf{w}_t - \frac{1}{2}\alpha\nabla_{\mathbf{w}_t}\left[v_{\pi_{\boldsymbol{\theta}}}(S_t) - \hat{v}(S_t, \mathbf{w}_t)\right]^2 \\ &= \mathbf{w}_t + \alpha\left[v_{\pi_{\boldsymbol{\theta}}}(S_t) - \hat{v}(S_t, \mathbf{w}_t)\right]\nabla_{\mathbf{w}}\hat{v}(S_t, \mathbf{w}_t) \quad \text{(applying chain rule of calculus).} \end{aligned} \tag{23}$$

Equation 23 assumes that $v_{\pi_{\boldsymbol{\theta}}}(S_t)$ is known, which is not. In fact, learning an approximation function would be futile if we knew the true state-value. Hence, to be of practical interest without breaking the convergence guarantees, we need to substitute $v_{\pi_{\boldsymbol{\theta}}}(S_t)$ in the equation by an unbiased noisy estimate U_t, i.e., $\mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left[U_t \mid S_t = s\right] = v_{\pi_{\boldsymbol{\theta}}}(s)$.

Given that the agent interacts in the environment according to $\pi_{\boldsymbol{\theta}}$ and that in those conditions the true value of a state is the expected value of the return following it, then G_t is by definition an unbiased estimate of $v_{\pi_{\boldsymbol{\theta}}}(S_t)$. As a consequence, we can safely assume $U_t \doteq G_t$, resulting in the following update equation:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\left[G_t - \hat{v}(S_t, \mathbf{w}_t)\right]\nabla_{\mathbf{w}}\hat{v}(S_t, \mathbf{w}_t).$$

4.10 The REINFORCE Algorithm with Baseline

Algorithm 4 presents the pseudo-code of the REINFORCE algorithm incorporating an estimated state-value function as a baseline. This version closely resembles the original REINFORCE algorithm without a baseline. In the baseline version, the algorithm updates the parameters of both the policy and state-value functions at each iteration. Notably, the algorithm employs two distinct learning rates: $\alpha^{\boldsymbol{\theta}}$ for the policy function and $\alpha^{\mathbf{w}}$ for the state-value approximation function.

Algorithm 4 Episodic REINFORCE Algorithm with baseline and decaying learning rate

Input: a differentiable policy parameterization $\pi(a \mid s, \boldsymbol{\theta})$,
Input: a differentiable state-value function parameterization $\hat{v}(s, \mathbf{w})$,
Parameters: step size $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$, discount factor $0 \leq \gamma \leq 1$, episode length T
Parameters: learning decay rates, $0 < \tau_{\boldsymbol{\theta}} \leq 1$ and $0 < \tau_{\mathbf{w}} \leq 1$, decay interval, $\delta t > 0$

Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^d$ (e.g., randomly)
Initialize state-value weights $\mathbf{w} \in \mathbb{R}^m$ (e.g., randomly)
Initialize global time step, $n \leftarrow 0$
Loop forever:
  // Generating an episode (policy rollout)
  Observe initial state S_0
  For each time step $t = 0, 1, \ldots, T-1$:
    Sample action from policy, $A_t \sim \pi(\cdot \mid S_t, \boldsymbol{\theta})$
    Take action A_t, observe reward R_{t+1} and next state S_{t+1}
  // Policy update based on episode return
  For each step of the episode $t = 0, 1, \ldots, T-1$:
    $n \leftarrow n + 1$
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\delta \leftarrow G - \hat{v}(S_t, \mathbf{w})$
    $\alpha^{\boldsymbol{\theta}} \leftarrow \alpha_0^{\boldsymbol{\theta}} \tau_{\boldsymbol{\theta}}^{\frac{n}{\delta t}}$
    $\alpha^{\mathbf{w}} \leftarrow \alpha_0^{\mathbf{w}} \tau_{\mathbf{w}}^{\frac{n}{\delta t}}$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t, \mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t \mid S_t, \boldsymbol{\theta})$

4.11 The Softmax Function

This section presents the *softmax* function, which will be used in the following section to transform action preferences onto action probabilites.

Formally, the *softmax* function converts a vector of n real numbers, $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$, into a normalized vector that represents a probability distribution. Specifically, each element of the resulting vector lies within the interval $(0, 1)$, $\tilde{\mathbf{x}} = (\tilde{x}_1, \ldots, \tilde{x}_n) \in (0, 1)^n$, and the elements sum to 1, ensuring that $\sum_{k=1}^{n} \tilde{x}_k = 1$. By denoting the i-th element of any vector $\mathbf{a}$ as $\mathbf{a}[i]$, the softmax function is applied element-wise as follows:

$$\text{softmax}(\mathbf{x})[i] = \frac{e^{\mathbf{x}[i]}}{\sum_{j=1}^{n} e^{\mathbf{x}[j]}}, \quad \forall\, 1 \leq i \leq n. \tag{24}$$

As an example, consider the vector $\mathbf{x} = (3.0, -1.0, 0.1)$. To apply the softmax function, we first compute the exponentials of each element: $e^{3.0} \approx 20.086$, $e^{-1.0} \approx 0.368$, and $e^{0.1} \approx 1.105$. The sum of these exponentials is $20.086 + 0.368 + 1.105 \approx 21.559$. The softmax values are then calculated as:

$$\text{softmax}(\mathbf{x})[1] = \frac{20.086}{21.559} \approx 0.932, \quad \text{softmax}(\mathbf{x})[2] = \frac{0.368}{21.559} \approx 0.017, \quad \text{softmax}(\mathbf{x})[3] = \frac{1.105}{21.559} \approx 0.051.$$

Thus, by applying the softmax function, the vector $\mathbf{x} = (3.0, -1.0, 0.1)$ is transformed into the probability distribution $\tilde{\mathbf{x}} = \text{softmax}(\mathbf{x}) = (0.932, 0.017, 0.051)$.

4.12 Policy Parameterization for Discrete Actions

Until now, we have not paid too much attention to the exact form of the policy function π, as long as it is differentiable and defines a probability distribution over actions.

The fact that π needs to be differentiable and define a probability distribution over actions imposes some constraints on its specific implementation. Imagine that π is a linear function of a set of state features (i.e., a higher-level state representation). The policy gradient update step may push and pull π such that it is no longer a probability distribution. For instance, it could happen that the policy gradient update step would discourage a bad action to the point of assigning a negative probability.

For discrete actions spaces, we can define a parameterized numerical preferences function $h : \mathcal{A} \times \mathcal{S} \times \mathbb{R}^d \to [0, 1]$. This function is not constrained to define a probability distribution. It only needs to ensure that higher numerical preferences are asserted to actions that are to be selected with higher probability. These preferences are then plugged in into a *softmax* function to define the policy function π, ensuring that it defines a proper probability distribution, whatever the real-valued preferences are are asserted to state-action pairs:

$$\pi(a \mid s, \boldsymbol{\theta}) \doteq \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_{b \in \mathcal{A}} e^{h(s,b,\boldsymbol{\theta})}}. \tag{25}$$

Note that policy parameters $\boldsymbol{\theta}$ parameterize function $h(\cdot, \cdot, \boldsymbol{\theta})$. Hence, the policy gradient with respect to $\boldsymbol{\theta}$, $\nabla_{\boldsymbol{\theta}} \ln \pi(\cdot \mid \cdot, \boldsymbol{\theta})$, needs to be propagated through h. Hence, to derive $\nabla_{\boldsymbol{\theta}} \ln \pi(\cdot \mid \cdot, \boldsymbol{\theta})$, we first take the logarithm of both sides in Equation 25:

$$\begin{aligned}
\ln \pi(a|s, \theta) &= \ln \frac{e^{h(s,a,\theta)}}{\sum_{b \in \mathcal{A}} e^{h(s,b,\theta)}} \quad \text{(take the logarithm in both sides)} \\
&= \ln e^{h(s,a,\theta)} - \ln \sum_{b \in \mathcal{A}} e^{h(s,b,\theta)} \quad \text{(because } \ln x/y = \ln x - \ln y \\
&= h(s, a, \theta) - \ln \sum_{b \in \mathcal{A}} e^{h(s,b,\theta)} \quad \text{(because } \ln e^x = x) \qquad (26)
\end{aligned}$$

From Equation 26, we take the gradient of $\ln \pi(a|s, \boldsymbol{\theta})$ with respect to $\boldsymbol{\theta}$:

$$
\begin{aligned}
\nabla_{\boldsymbol{\theta}} \ln \pi(a|s,\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} h(s,a,\boldsymbol{\theta}) - \nabla_{\boldsymbol{\theta}} \ln \sum_{b\in\mathcal{A}} e^{h(s,b,\boldsymbol{\theta})} \quad \text{(because } (f(x)+g(x))' = f'(x)+g'(x)\text{)} \\
&= \nabla_{\boldsymbol{\theta}} h(s,a,\boldsymbol{\theta}) - \frac{\nabla_{\boldsymbol{\theta}} \sum_{b\in\mathcal{A}} e^{h(s,b,\boldsymbol{\theta})}}{\sum_{b\in\mathcal{A}} e^{h(s,b,\boldsymbol{\theta})}} \quad \text{(because } (\ln f(x))' = f'(x)/f(x)\text{)} \\
&= \nabla_{\boldsymbol{\theta}} h(s,a,\boldsymbol{\theta}) - \frac{\sum_{b\in\mathcal{A}} \nabla_{\boldsymbol{\theta}} e^{h(s,b,\boldsymbol{\theta})}}{\sum_{b\in\mathcal{A}} e^{h(s,b,\boldsymbol{\theta})}} \quad \text{(because } (f(x)+g(x))' = f'(x)+g'(x)\text{)} \\
&= \nabla_{\boldsymbol{\theta}} h(s,a,\boldsymbol{\theta}) - \frac{\sum_{b\in\mathcal{A}} \nabla_{\boldsymbol{\theta}} h(s,b,\boldsymbol{\theta}) e^{h(s,b,\boldsymbol{\theta})}}{\sum_{b\in\mathcal{A}} e^{h(s,b,\boldsymbol{\theta})}} \quad \text{(because } [f(g(x))]' = f'(g(x))g'(x)\text{)} \\
&= \nabla_{\boldsymbol{\theta}} h(s,a,\boldsymbol{\theta}) - \sum_{b\in\mathcal{A}} \nabla_{\boldsymbol{\theta}} h(s,b,\boldsymbol{\theta}) \pi(b|s,\boldsymbol{\theta}) \quad \text{(substituting Equation 25).} \qquad (27)
\end{aligned}
$$

4.13 Linear Policies and State-Value Functions

In this section, we derive the simplest parameterized preference function $h(s,a,\boldsymbol{\theta})$. Specifically, we define the preference for taking action a in state s as a linear combination of features of both a and s:

$$h(s,a,\boldsymbol{\theta}) = \boldsymbol{\theta}^T \mathbf{x}(s,a), \qquad (28)$$

where $\mathbf{x} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^d$ is the d-dimensional vector of features for the given station-action pair. We need d learnable parameters to solve this task, $|\boldsymbol{\theta}| = d$.

The feature vector could be as simple as the state itself, $\mathbf{x}(s,a) \doteq s$, provided it is a real-valued vector. More generally, a feature is a representation of the state-action pair that is believed to be useful for solving the task at hand. For example, if the agent's task is to approach a given goal g, a relevant feature might be the distance from the current state location to the goal location. Features capture the non-linear aspects of the task that are essential and would otherwise be missed by a linear policy. Moreover, features allow us to incorporate prior domain knowledge into the reinforcement learning problem. Without features, the agent would have to learn this knowledge from scratch, necessitating more complex policy functions, which would, in turn, slow down the learning process. The caveat of using features is that often it is difficult to do the right *feature engineering*.

Now that we have defined the function h, we can compute its gradient with respect to the learnable parameters. In the linear case, this computation is straightforward:

$$\nabla_{\boldsymbol{\theta}} h(s,a,\boldsymbol{\theta}) = \mathbf{x}(s,a). \qquad (29)$$

By substituting Equation 29 into Equation 27, we obtain the gradient of the logarithm of the policy with respect to $\boldsymbol{\theta}$. This gradient can then be used in a policy gradient algorithm to update the policy parameters, as in Step 14 of Algorithm 3:

$$\nabla_{\boldsymbol{\theta}} \ln \pi(a|s,\boldsymbol{\theta}) = \mathbf{x}(s,a) - \sum_{b\in\mathcal{A}} \mathbf{x}(s,b)\pi(b|s,\boldsymbol{\theta}). \qquad (30)$$

How do we evaluate Equation 30? For a given state s and action a, we construct the feature vector by evaluating $\mathbf{x}(s,a)$ (e.g., the distance between s and a goal g). Each time we need to evaluate the policy function, we first compute $h(s,a,\boldsymbol{\theta})$ for state s, action a, and the current values of the policy parameters $\boldsymbol{\theta}$ (Equation 28). The resulting numerical preference is then fed into Equation 25, producing a probability. This probability can finally be substituted into Equation 30 to obtain the gradient.

Limitations

Linear policies are primarily of educational interest due to their limited expressiveness, which makes them unsuitable for complex problems that require non-linear dependencies on the feature vector. This limitation also extends to linear state-value function approximations used as baselines.

Depending on the specific problem, learning the policy function can vary in difficulty compared to learning the state-value function. One reason learning policies can be easier than state-value functions is that policies directly map states to actions, making the learning process more straightforward in environments where the optimal action for a given state is relatively clear. In contrast, state-value functions must estimate the expected return for a state under a given policy, which requires accurately

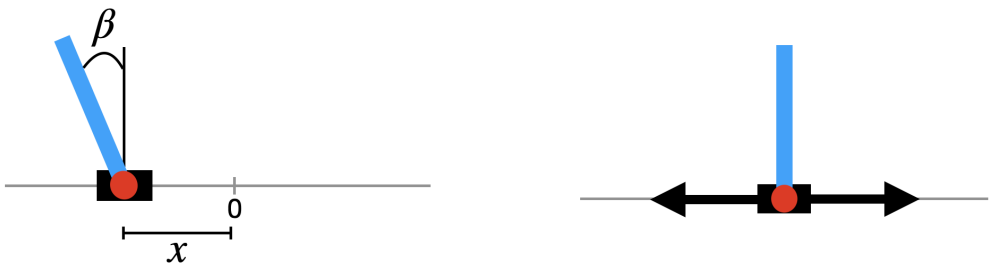
capturing long-term future rewards. This can be more challenging, especially in environments with high variance in rewards or delayed consequences of actions.

When the state-value function is harder to learn, relying on a linear model becomes particularly restrictive, potentially hindering the effective learning of the policy. This issue arises because the sampled return is compared against an erroneous expected return, leading to inaccurate evaluations and suboptimal policy updates. Inaccurate state-value estimates can misguide the learning process, causing the policy to deviate from the optimal path. Consequently, a more expressive model, capable of capturing non-linear relationships, is often necessary to achieve better performance in complex problems.

Example: The CartPole Problem

The CartPole problem, instantiated by OpenAI as CartPole-v1[1], is a classic problem in reinforcement learning that involves balancing a pole on a moving cart, subject to the dynamics imposed by the laws of physics. The goal is to keep a pole, attached to a cart via a passive (non-actuated) joint, balanced upright by moving the cart either to the left or to the right along a frictionless track. This is a typical toy problem often used to test and benchmark reinforcement learning algorithms. Despite its simplicity, it includes the non-linear dynamics involved in balancing the pole. Figure 15 depicts the problem, including its state and action spaces.

The state is represented by a 4-dimensional vector, $s \doteq (x, x', \beta, \beta')$ including the real-valued position of the cart on the track, x, the real-valued linear velocity of the cart, x', the real-valued angle of the pole with the vertical, β, and the real-valued angular velocity of the pole, β'. The agent can execute one of two discrete actions, $\mathcal{A} = \{\text{left}, \text{right}\}$: applying a constant force on the cart to accelerate it to the left; applying a constant force on the cart to accelerate it to the right. The agent earns a reward of $+1$ for every time step until the episode terminates. Hence, the longer the episode, the larger the reward. The episode terminates when any of the following conditions occur: the pole falls past ± 12 degrees from the vertical or the cart moves beyond ± 2.4 units from the center; or a maximum of 500 time steps is reached.



(a) State (b) Actions

Figure 15: State and actions of the CartPole problem. The cart, the pole, and the passive joint are represented in black, blue, and respectively. Left: episode terminating because the position x and angle β off the limits. Right: the cart in the "desired" position and the pole in the "desired" angle, with the black arrows representing the two possible discrete actions (forces).

To instantiate the REINFORCE algorithm to this problem we need to make some decisions. We define the feature vector as simply the state, $\mathbf{x}(s, a) \doteq s$, thus independent of the action. We define the action preferences function as a linear function of the state (Equation 28), using separate weights for stating the preferences for the left action and the preferences for the right action, $\boldsymbol{\theta} = \boldsymbol{\theta_1} \cup \boldsymbol{\theta_2}$:

$$h(s, a, \boldsymbol{\theta}) = \begin{cases} \boldsymbol{\theta_1^T} s & \text{if } a = \text{left} \\ \boldsymbol{\theta_2^T} s & \text{if } a = \text{right} \end{cases}. \tag{31}$$

As in Equation 26 and based on Equation 31, we derive the policy logarithm, but this time separately for the two actions (for the sake of clarity):

[1] https://www.gymlibrary.dev/environments/classic_control/cart_pole/

$$\ln \pi(\text{left} \quad | \, s, \boldsymbol{\theta}) = \ln \frac{e^{h(s,\text{left},\boldsymbol{\theta})}}{e^{h(s,\text{left},\boldsymbol{\theta})} + e^{h(s,\text{right},\boldsymbol{\theta})}} = \ln \frac{e^{\boldsymbol{\theta_1^T} s}}{e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s}} = \boldsymbol{\theta_1^T} s - \ln(e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s}) \tag{32}$$

$$\ln \pi(\text{right} \mid s, \boldsymbol{\theta}) = \ln \frac{e^{h(s,\text{right},\boldsymbol{\theta})}}{e^{h(s,\text{left},\boldsymbol{\theta})} + e^{h(s,\text{right},\boldsymbol{\theta})}} = \ln \frac{e^{\boldsymbol{\theta_2^T} s}}{e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s}} = \boldsymbol{\theta_2^T} s - \ln(e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s}). \tag{33}$$

Although the action preferences are defined by different sets of parameters, both influence the policy to ensure the latter defines a proper probability distribution (i.e., normalizes the action preferences to sum up to 1 and to ensure that none is negative or above 1). The next step is to take the gradients of the logarithms, as we have done in Equation 30, but now with respect to the different sets of parameters (recall that $\pi(\text{left} \mid s, \boldsymbol{\theta}) = 1 - \pi(\text{right} \mid s, \boldsymbol{\theta})$):

$$\nabla_{\boldsymbol{\theta_1}} \ln \pi(a \mid s, \boldsymbol{\theta}) = \begin{cases} -\pi(\text{left} \mid s, \boldsymbol{\theta}) s & \text{if } a = \text{right} \\ \pi(\text{right} \mid s, \boldsymbol{\theta}) s & \text{if } a = \text{left} \end{cases}, \tag{34}$$

$$\nabla_{\boldsymbol{\theta_2}} \ln \pi(a \mid s, \boldsymbol{\theta}) = \begin{cases} -\pi(\text{right} \mid s, \boldsymbol{\theta}) s & \text{if } a = \text{left} \\ \pi(\text{left} \mid s, \boldsymbol{\theta}) s & \text{if } a = \text{right} \end{cases}. \tag{35}$$

Derivations (for the interested reader)

$$\begin{aligned}
\nabla_{\boldsymbol{\theta_1}} \ln \pi(\text{left} \mid s, \theta) &= \nabla_{\boldsymbol{\theta_1}} \left[\boldsymbol{\theta_1^T} s - \ln(e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s}) \right] \\
&= s - \nabla_{\boldsymbol{\theta_1^T}} \ln(e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s}) \\
&= s - \frac{\nabla_{\boldsymbol{\theta_1^T}} (e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s})}{e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s}} \\
&= s - \frac{s e^{\boldsymbol{\theta_1^T} s}}{e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s}} \\
&= s - s\pi(\text{left} \mid s, \theta) \\
&= s(1 - \pi(\text{left} \mid s, \theta)) \\
&= \pi(\text{right} \mid s, \theta) s
\end{aligned}$$

$$\begin{aligned}
\nabla_{\boldsymbol{\theta_2}} \ln \pi(\text{right} \mid s, \theta) &= \nabla_{\boldsymbol{\theta_2}} \left[\boldsymbol{\theta_2^T} s - \ln(e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s}) \right] \\
&= s - \nabla_{\boldsymbol{\theta_2^T}} \ln(e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s}) \\
&= s - \frac{\nabla_{\boldsymbol{\theta_2^T}} (e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s})}{e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s}} \\
&= s - \frac{s e^{\boldsymbol{\theta_2^T} s}}{e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s}} \\
&= s - s\pi(\text{right} \mid s, \theta) \\
&= s(1 - \pi(\text{right} \mid s, \theta)) \\
&= \pi(\text{left} \mid s, \theta) s
\end{aligned}$$

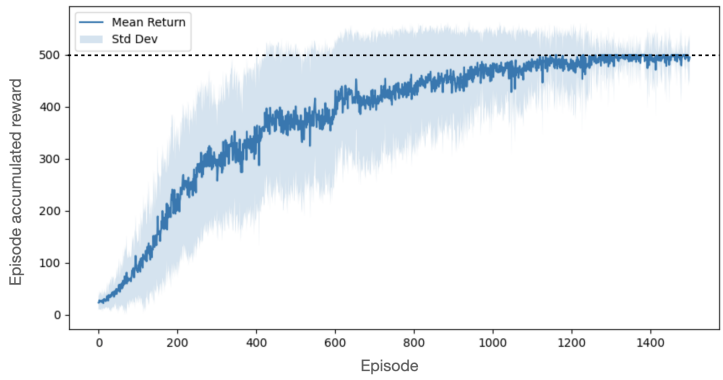
$$\begin{aligned}
\nabla_{\boldsymbol{\theta_2}} \ln \pi(\text{left} \mid s, \theta) &= \nabla_{\boldsymbol{\theta_2}} \left[\boldsymbol{\theta_1^T} s - \ln(e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s}) \right] \\
&= -\nabla_{\boldsymbol{\theta_2^T}} \ln(e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s}) \\
&= -\frac{\nabla_{\boldsymbol{\theta_2^T}} (e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s})}{e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s}} \\
&= -\frac{s e^{\boldsymbol{\theta_2^T} s}}{e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s}} \\
&= -\pi(\text{right} \mid s, \theta) s
\end{aligned}$$

$$\begin{aligned}
\nabla_{\boldsymbol{\theta_1}} \ln \pi(\text{right} \mid s, \theta) &= \nabla_{\boldsymbol{\theta_1}} \left[\boldsymbol{\theta_2^T} s - \ln(e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s}) \right] \\
&= -\nabla_{\boldsymbol{\theta_1^T}} \ln(e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s}) \\
&= -\frac{\nabla_{\boldsymbol{\theta_1^T}} (e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s})}{e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s}} \\
&= -\frac{s e^{\boldsymbol{\theta_1^T} s}}{e^{\boldsymbol{\theta_1^T} s} + e^{\boldsymbol{\theta_2^T} s}} \\
&= -\pi(\text{left} \mid s, \theta) s
\end{aligned}$$

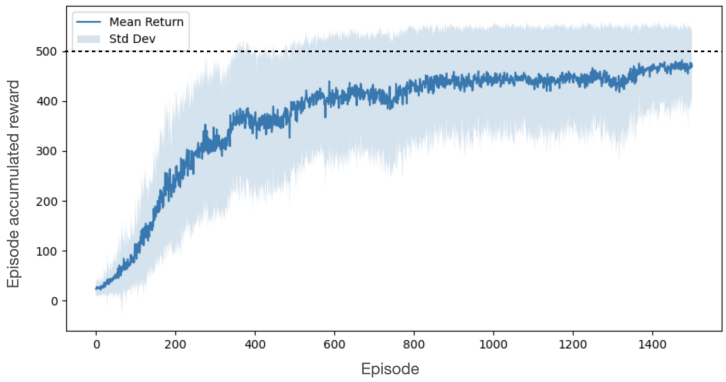
Results

Figure 16 illustrates the learning progress of the REINFORCE algorithm with decaying learning rate (but without baseline), using a base learning rate of $\alpha_0 = 0.001$, for learning decay rates $\tau = 0.85$ (and $\delta_t = 100$) and $\tau = 1.0$, in the CartPole-v1 problem. The figure shows the average and standard deviation of the accumulated (undiscounted) rewards per learning episode, averaged over 30 runs.

Using an exponentially decaying learning rate (i.e., $\tau = 0.85$) enables the accumulated reward to increase steadily, approaching the maximum possible reward of 500 more closely than without an adaptive learning rate (i.e., $\tau = 1.0$). The plot also shows a large standard deviation, reflecting significant variability in results across different runs, a common characteristic of reinforcement learning. This variability partly arises from different initial conditions set by various seeds for the pseudo-random



(a) $\tau = 0.85$ (and $\delta_t = 100$)



(b) $\tau = 1.0$

Figure 16: REINFORCE algorithm with decaying learning rate (but without baseline), using a base learning rate of $\alpha_0 = 0.001$, in the CartPole-v1 problem. The plots present the average and standard deviation of accumulated reward (undiscounted), per learning episode, over 30 runs.

number generator, leading to varying learning trajectories and potentially causing the agent to reach local maxima instead of the global maximum.

The observed variability is also due to the exploratory nature of the learning algorithm, which selects actions by stochastically sampling the policy at each instant. This means the agent does not always perform optimally in the sense of greedily exploiting the best action. However, running the learned policy greedily, i.e., $A_t = \arg\max_a \pi(a|S_t, \boldsymbol{\theta})$, for 30 episodes after each of the 30 runs consistently achieved the perfect accumulated reward of 500. Thus, the algorithm successfully learned the optimal policy. Nevertheless, to continue learning and adapt to changing environments, it is necessary to accept the cost of occasional suboptimal exploratory behavior.

4.14 Non-Linear Policies and State-Value Functions

In the previous section, we modeled action preference functions $h(s, a, \boldsymbol{\theta})$ as a linear function of the state feature vector $\mathbf{x}(s, a)$. However, linear models often fall short in capturing the complex relationships between state features and action preferences that are crucial for solving sophisticated tasks. Additionally, relying on linear models necessitates the manual design of feature spaces, which can be labor-intensive and may require significant domain expertise.

To address these limitations, a more powerful approach is to utilize *deep* Artificial Neural Networks (ANNs) to model action preferences directly as non-linear functions of the raw state vector s. This method introduces non-linearity into the policy representation, effectively allowing the model to learn complex patterns from the state space without the need for extensive feature engineering. By leveraging ANNs, particularly those with multiple layers, we can build flexible and highly expressive models that automatically learn to represent intricate dependencies between states and actions.

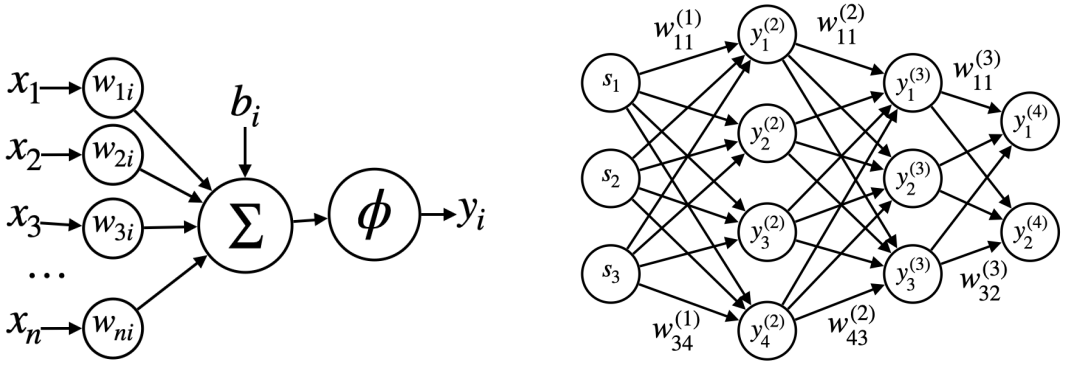
Multilayer Perceptron (MLP)

A commonly used ANN architecture for policy and state-value modeling is the Multilayer Perceptron (MLP). An MLP is composed of several simple information processing units, known as artificial neurons.

Each neuron generates as output a scalar resulting from a simple non-linear operation on the neuron's inputs. Formally, a neuron i applies a non-linear transformation $\phi : \mathbb{R} \to \mathbb{R}$, also known as an *activation function*, to the scalar bias $b_i \in \mathbb{R}$ plus the weighted sum of its inputs $\{x_j\} \in \mathbb{R}$ (see Figure17a):

$$y_i = \phi \left(b_i + \sum_j x_j w_{ji} \right),$$

where $w_{ji} \in \mathbb{R}$ is the weight indicating how much the neuron's input x_j influences its output. Learning occurs by adapting the neuron's bias and weights so the neuron's output matches the intended value.



(a) (b)

Figure 17: Diagram of an artificial neuron (Left) and of an MLP (Right). The neurons in the MLP are represented by their outputs and the layers have dimensions $n_1 = 3, n_2 = 4, n_3 = 3, n_4 = 2$.

In an MLP, neurons are organized in layers (see Figure17b). Let us denote the number of neurons in a given layer l by n_l, where the first layer is $l = 1$ and the last layer is $l = L$. The first and last layers are commonly known as *input* and *output* layers, respectively. All other intermediate layers are commonly known as *hidden layers*. The neurons in the first layer, $l = 1$, directly output the state vector s, that is, each neuron i outputs the i-th element of s, denoted by s_i, where $s = (s_i)_i$. All neurons in subsequent layers, $l > 1$, receive as input the outputs of all neurons in the preceding layer, $l - 1$. In addition, all neurons in layer $l > 1$ employ the same activation function $\phi^{(l)}$. We redefine the neuron equation by explicitly representing the dependence on the layer l with the superscript (l), as well as on the state s and the vector that encompasses all weights and biases of the MLP, $\boldsymbol{\theta}$:

$$y_i^{(l)}(s, \boldsymbol{\theta}) = \begin{cases} s_i & \text{if } l = 1, \\ \phi^{(l)} \left(b_i^{(l)} + \sum_j w_{ji}^{(l-1)} y_j^{(l-1)}(s, \boldsymbol{\theta}) \right) & \text{if } l > 1. \end{cases} \tag{36}$$

The inclusion of the non-linear activation function is vital for the MLP's ability to model non-linear relationships in the state space. Non-linearities allow the network to capture complex dynamics and decision boundaries that are critical for effective learning and generalization. Two commonly used activation functions are the Rectified Linear Unit (ReLU) and the hyperbolic tangent (tanh) function.

In its basic formulation, the ReLU function is linear for positive values and zero otherwise, ensuring a large derivative when positive while simultaneously introducing the required non-linearity. The ReLU activation function is typically employed in all layers except the last one:

$$\phi^{(l)}(x) = \begin{cases} \max(0, x) & \text{if } l < L, \\ x & \text{if } l = L. \end{cases}$$

The hyperbolic tangent activation function, *tanh*, maps input values to the range $[-1, 1]$, introducing non-linearity while maintaining the ability to model both positive and negative relationships. It is defined as:

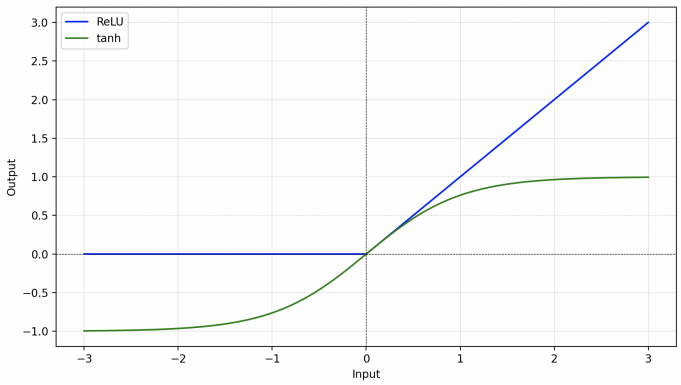


Figure 18: Tanh and ReLu activation functions.

$$\phi(x) = \tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

The *tanh* function is particularly important in reinforcement learning when outputs need to be bounded within a specific range, such as when actions are constrained between $[-1, 1]$. Additionally, the zero-centered output of the *tanh* function can lead to faster convergence during training by improving the symmetry of gradient updates.

Figure 18 plots both *tanh* and ReLU activation functions. The choice between ReLU and *tanh* often depends on the specific requirements of the problem. While ReLU is computationally efficient and alleviates the vanishing gradient problem, the bounded nature of *tanh* makes it well-suited for tasks where the action space or value representation is naturally constrained. In practice, these activation functions may be combined in different parts of the network to exploit their respective advantages. The interested reader can find a thorough introduction to deep learning and ANNs in [1].

MLP for Policy Modeling

An MLP can be used for modeling a policy in REINFORCE by considering the outputs of all neurons in the last layer L as a vector that defines the preferences over the n_L possible actions, one neuron per action, given a state s and the vector that encompasses all weights and biases of the MLP, $\boldsymbol{\theta}$:

$$\mathbf{y}^{(L)}(s, \boldsymbol{\theta}) = (y_1^{(L)}(s, \boldsymbol{\theta}), y_2^{(L)}(s, \boldsymbol{\theta}), \ldots, y_{n_L}^{(L)}(s, \boldsymbol{\theta})).$$

The preferences vector is transformed into a probability distribution over the n_L actions by applying the softmax function (as in Equation 25), resulting in the following policy function:

$$\pi(a|s, \boldsymbol{\theta}) \doteq \text{softmax}(\mathbf{y}^{(L)}(s, \boldsymbol{\theta}))[a],$$

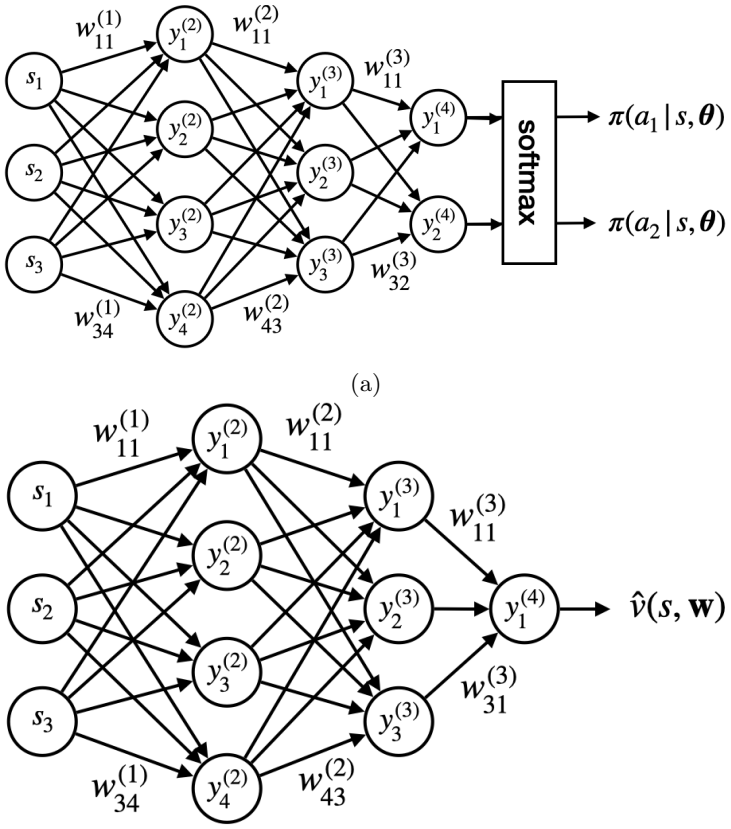
where $\text{softmax}(\mathbf{y}^{(L)}(s, \boldsymbol{\theta}))[a]$ indicates the probability assigned to action a (see Section 4.11). Figure 19a depicts an example MLP for policy function modeling.

MLP for State-Value Function Modeling

We can also use an MLP to model a state-value function to be used as a baseline in REINFORCE. The MLP for state-value functions is often similar to the one used for policy modeling, with one important difference in the last layer. The last layer consists of a single neuron, $n_L = 1$, as the goal of this MLP is to output a scalar value representing the estimated value of the state s, not a probability:

$$\hat{v}(s, \mathbf{w}) \doteq y_1^{(L)}(s, \mathbf{w}),$$

where $\mathbf{w}$ is the set of weights and biases that parameterizes the MLP (differentiated from the parameters used for the policy function MLP, $\boldsymbol{\theta}$). Figure 19b depicts an example MLP for state-value function modeling.



(a)

(b)

Figure 19: Diagram of an MLP for policy (Left) and state-value function (Right) modeling.

Training the MLP

Training a neural network is a time-dependent operation. Hence, all weights and biases are time-indexed, they evolve as training unfolds. This dependence is hereafter made explicit by including the subscript t.

MLP layers operate as multi-input multi-output functions feeding one another in sequence. In this sense, an MLP is a composite function, where each layer i represents a distinct function f_i (represented by Equation 36) with its own set of learnable parameters: weights and biases.

For instance, a three-layer network for modeling a state-value function can be expressed as a composition of three functions that take the state vector s as input and produce the estimated value function according to the learnable parameters $\mathbf{w}$:

$$\hat{v}(s, \mathbf{w}) \doteq f_3(f_2(f_1(s, \mathbf{w}_t^{(1)}), \mathbf{w}^{(2)}), \mathbf{w}^{(3)}), \quad \text{where } \mathbf{w} = \mathbf{w}^{(1)} \cup \mathbf{w}^{(2)} \cup \mathbf{w}^{(3)}.$$

As we have seen, training a parametric state-value function amounts to updating each parameter in the opposite direction of the *mean squared error* function's gradient with respect to the parameter, that is, performing a *gradient descent*. It is common to refer to this error function as the *loss function*. This is a minimization procedure that aims to bring the function's state-value prediction close to the actual state value. Incapable of knowing beforehand the actual state value, the best we can do is to use sample-based estimates. This can be done by running the agent in the environment from state S_t according to the current policy π for a whole episode and considering the obtained return G_t as a sample-based estimate of the state value. In this case, the loss is defined as:

$$\mathcal{L}_t^{\hat{v}}(\mathbf{w}) = \left(G_t - \hat{v}(S_t, \mathbf{w})\right)^2. \tag{37}$$

Given that $\hat{v}(S_t, \mathbf{w})$ is a composite function, the gradient of the loss function $\nabla_{\mathbf{w}} \left(G_t - \hat{v}(S_t, \mathbf{w})\right)^2$, is also a composite function. Hence, computing its gradient with respect to $w^{(k)} \in \mathbf{w}^{(k)}$, can be done by applying the chain rule of calculus, considering $y_k = f_k(\cdot, \cdot)$:

$$\frac{d\mathcal{L}_t^{\hat{v}}}{dw^{(1)}} = \frac{d\mathcal{L}_t^{\hat{v}}}{dy_3} \cdot \frac{dy_3}{dy_2} \cdot \frac{dy_2}{dy_1} \cdot \frac{dy_1}{dw^{(1)}},$$
$$\frac{d\mathcal{L}_t^{\hat{v}}}{dw^{(2)}} = \frac{d\mathcal{L}_t^{\hat{v}}}{dy_3} \cdot \frac{dy_3}{dy_2} \cdot \frac{dy_2}{dw^{(2)}},$$
$$\frac{d\mathcal{L}_t^{\hat{v}}}{dw^{(3)}} = \frac{d\mathcal{L}_t^{\hat{v}}}{dy_3} \cdot \frac{dy_3}{dw^{(3)}}.$$

Training an MLP for modeling a policy follows a similar procedure, the only difference being the way the function to be optimized is defined, which results in the need to include the logarithm of the softmax function in the chain rule. Concretely, the parameters are updated in the direction of the gradient of $G_t \ln \pi(A_t|S_t, \boldsymbol{\theta})$ (G_t can be subtracted by a baseline). That is, the goal of the optimization is to make more probable those actions that are correlated with higher returns by performing *gradient ascent* on $G_t \ln \pi(A_t|S_t, \boldsymbol{\theta})$. As the goal is maximizing $G_t \ln \pi(A_t|S_t, \boldsymbol{\theta})$, we define the following *objective function* to be optimized, instead of a loss function:

$$\mathcal{L}_t^{\pi}(\boldsymbol{\theta}_t) = G_t \ln \pi(A_t|S_t, \boldsymbol{\theta}) = G_t \ln \text{softmax}(\mathbf{y}^{(L)}(S_t, \boldsymbol{\theta}))[A_t],$$

where G_t is the return obtained by the agent when starting in state S_t, performing action A_t, and then following policy π thereafter for the whole episode.

If we want to optimize $\mathcal{L}_t^{\pi}(\boldsymbol{\theta})$ with *gradient descent*, which is more typical in neural networks training toolkits, we just need to negate the objective function in order to transform it into a loss function: $\mathcal{L}_t^{\pi}(\boldsymbol{\theta}) = -G_t \ln \pi(A_t|S_t, \boldsymbol{\theta})$. It is common for both objective functions and loss functions to be referred to simply as loss functions, with the distinction between using gradient ascent or descent being implicitly understood based on the function's intended goal. Here, we choose to distinguish between objective functions and loss functions to avoid ambiguity, explicitly recognizing their differing roles and optimization directions.

As the network's complexity increases, finding a closed-form solution for these gradients becomes increasingly difficult. Instead, the backpropagation algorithm is used to efficiently compute these gradients by iterating backward through the network, layer by layer. Backpropagation avoids redundant calculations of intermediate terms by storing partial derivatives, making it highly suitable for training deep networks.

Improving the training of the MLP

While simple gradient descent can update the parameters, more sophisticated optimization algorithms like Adam are often preferred to improve training efficiency and convergence. Adam, short for Adaptive Moment Estimation, is a popular optimization algorithm in deep learning that combines the benefits of momentum and RMSprop. Momentum accelerates convergence by smoothing the gradient updates, while RMSprop adapts the learning rate for each parameter based on recent gradient magnitudes. Adam integrates these methods by using exponentially weighted moving averages of both the gradients' mean (first moment) and variance (second moment), allowing it to adapt the learning rate dynamically for each parameter, making it more robust to noisy or sparse data.

In training neural networks, a significant challenge is overfitting, where the model becomes overly complex and performs well on training data but fails to generalize to new, unseen data. Overfitting occurs when the model captures not just the underlying patterns but also the noise in the training set. To combat this, regularization techniques are used, with weight decay being one of the most effective and commonly applied methods. Weight decay, also known as L2 regularization, is implemented by adding a penalty term to the model's loss function that is proportional to the squared L2 norm of the model's weights. When dealing with an objective function (to be maximized), the penalty term is subtracted instead of added, reflecting the opposite optimization direction. Formally, if the original loss function is $\mathcal{L}(\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ represents the vector of model weights, the modified loss function with weight decay is:

$$\mathcal{L}_{regularised}(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|^2,$$

where λ is a hyperparameter that controls the strength of the regularization, and $\|\boldsymbol{\theta}\|^2$ represents the squared L2 norm of the weight vector. This penalty term discourages the model from assigning excessively large weights to any single feature, effectively keeping the weights small.

The benefits of weight decay go beyond just preventing overfitting. By constraining the weights, it stabilizes the training process, ensuring that the model does not develop excessively large weights, which could lead to unstable and erratic behavior during optimization. Additionally, by encouraging smaller weights, weight decay implicitly guides the model toward simpler solutions that are more likely to generalize well to new data. In essence, weight decay helps in producing models that are both robust and effective in making accurate predictions on unseen data.

In the training of an MLP, we can halt training when the rewards stabilize over a set of sequential episodes. This prevents overfitting to the specific trajectories encountered during training and ensures the model does not become overly specialized. Early stopping also saves computational resources by avoiding unnecessary training once the model has converged.

For stable and efficient learning, it is crucial that the inputs to the MLP are zero-centered and have unit variance. This ensures that each input dimension contributes equally to the learning process, preventing any single feature from disproportionately influencing the updates. To achieve this, the state vector $S_t = (S_{i,t})_i$ should be normalized before being fed into the MLPs. Typically, a normalized version of the state vector, $\tilde{S}_t = (\tilde{S}_{i,t})_i$, is defined as:

$$\tilde{S}_{i,t} = \frac{S_{i,t} - \mu_{i,t}}{\sigma_{i,t}}, \tag{38}$$

where $\mu_{i,t}$ and $\sigma_{i,t}$ represent the running mean and standard deviation, respectively, of the i-th component of the state vector up to time step t. These statistics are updated incrementally as the agent interacts with the environment, enabling the model to adjust to shifts in the data distribution over time.

Example 1: The CartPole Problem (revisited)

Let us now solve the CartPole-v1 problem using REINFORCE, this time employing an MLP as the policy function and another MLP as the state-value function (for baseline), as formulated in the previous sections.

Given that state in CartPole-v1 is represented by a 4-dimensional vector, $s \doteq (x, x', \theta, \theta')$, both MLPs have 4 neurons in the first layer, $n_1 = 4$. The MLP for the policy function has two hidden layers with 128 neurons each, $n_2 = n_3 = 128$, whereas the MLP for the value function has two hidden layers with fewer neurons , $n_2 = 32, n_3 = 24$. The output layer in the MLP for the policy function has two neurons, $n_4 = 2$, as many as the number of possible actions, $\mathcal{A} = \{\text{left}, \text{right}\}$. For improved learning, the MLP is fed with the normalized version of the state, $\tilde{s}$ (Equation 38).

The state value changes as the policy changes. Hence, the state-value function MLP must train faster than the policy function MLP. This is ensured by setting the learning rates for the policy and state-value functions MLPs as 1e-4 and 1e-2, respectively. The discount factor in both MLPs has been set to $\gamma = 0.99$. The hyperparameter controlling the strength of the weight decay regularization has been set to $\lambda = 0.02$. Training is early stopped as soon as we find a sequence of five episodes in which the maximum reward of 500 is achieved.

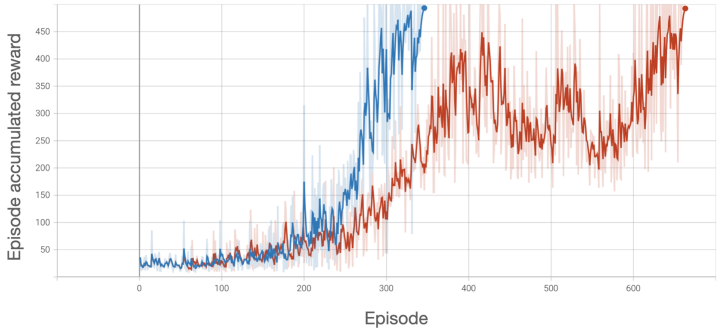
Figure 20 displays the accumulated reward (undiscounted) per learning episode for the CartPole-v1 problem, using the REINFORCE algorithm with MLPs for both the policy and state-value functions. The results demonstrate that the problem is successfully solved. Additionally, the figure highlights that incorporating the state-value function MLP as a baseline leads to faster and more stable learning compared to training without a baseline.

Example 2: The Lunar Lander Problem

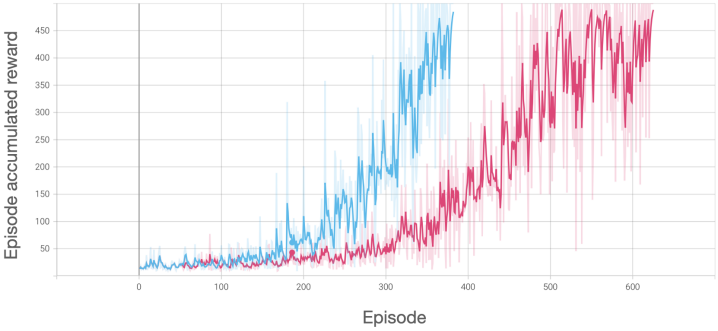
The Lunar Lander problem, instantiated by OpenAI as LunarLander-v2[2], is a well-known benchmark in reinforcement learning that involves safely landing a spacecraft on the surface of the moon (see Figure 21). The goal is to control the lander's descent using its main and side thrusters, ensuring a soft landing on the designated landing pad, while adhering to the dynamics imposed by the simulated environment. This problem encompasses challenges related to continuous control, delayed rewards, and the need for precision.

The state is represented by an 8-dimensional vector, $s \doteq (x, y, \dot{x}, \dot{y}, \theta, \dot{\theta}, l_c, r_c)$, where x and y denote the lander's horizontal and vertical positions, $\dot{x}$ and $\dot{y}$ are the horizontal and vertical velocities, θ is the angle of the lander relative to the vertical, and $\dot{\theta}$ is its angular velocity. The binary variables l_c and r_c indicate whether the left or right leg of the lander is in contact with the ground, respectively. The agent can

[2] `https://www.gymlibrary.dev/environments/box2d/lunar_lander/`



(a) Seed 100.



(b) Seed 200.

Figure 20: Smoothed accumulated reward (undiscounted) per learning episode for the CartPole-v1 problem, using the REINFORCE algorithm with MLPs for policy and state-value functions. The left and right plots correspond to two different random seeds, with unsmoothed data shown in lower opacity. Blue plots indicate the use of the state-value function MLP as a baseline, while red plots represent results without a baseline.

execute one of four discrete actions, $\mathcal{A} = \{\text{do nothing}, \text{fire left engine}, \text{fire main engine}, \text{fire right engine}\}$. Choosing to "do nothing" results in no thrust being applied. Firing the left engine generates a clockwise torque, while firing the right engine produces a counterclockwise torque. Firing the main engine applies an upward force, reducing the descent velocity and controlling the lander's vertical motion.

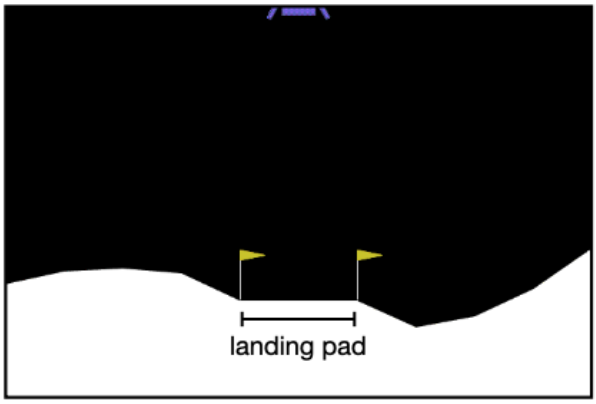
The lander earns approximately 100 to 140 points for successfully descending from the top of the screen to the landing pad and coming to a complete stop. Points are deducted if the lander drifts away from the landing pad. A crash results in an additional penalty of -100 points, while safely coming to rest grants an extra +100 points. The lander also earns +10 points for each leg that makes contact with the ground. Using the main engine incurs a cost of -0.3 points per frame, and using the side engines costs -0.03 points per frame. The problem is considered solved if the agent achieves a score of 200 points.

The lunar lander starts each episode positioned at the top center of the viewport, with a random initial force applied to its center of mass. The episode concludes under several conditions: if the lander crashes by making contact with its body in the moon's surface, if it moves outside the bounds of the viewport (specifically, if its x-coordinate exceeds the boundary), or if the lander does not move and does not collide with any other body.

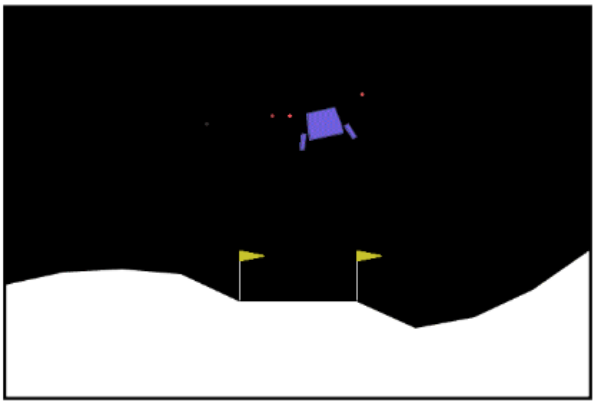
Figure 22 shows the accumulated reward (undiscounted) per learning episode for the LunarLander-v2 problem, using the REINFORCE algorithm with MLPs for both the policy and state-value functions, similar to the approach used for the CartPole-v1 problem and using the baseline. The results indicate that the problem is successfully solved, with rewards consistently approaching 300, though it requires significantly more training time compared to CartPole-v1. The plot also reveals that learning stagnated during several episodes before making a final push toward higher rewards.

5 On-Policy Actor-Critic Methods

In the previous section, we explored the foundational principles of policy gradient methods, culminating in the REINFORCE algorithm, which trains non-linear policies represented by artificial neural networks.



(a) Episode initial state.



(b) Episode's intermediate state.

Figure 21: The Lunar Lander problem. The objective is to safely land the spacecraft within the designated landing area, marked by the space between the yellow flags. The small dots emanating from the lander indicate the activation of its thrusters.

While REINFORCE laid the groundwork, it has limitations that hinder its effectiveness in solving more complex reinforcement learning problems.

In this section, we extend our understanding to the concepts underlying Proximal Policy Optimization (PPO), one of the most widely used and effective reinforcement learning algorithms. PPO is a *model-free*, *on-policy*, *actor-critic* method, which combines policy gradients with value-based estimators. This combination addresses the shortcomings of policy gradient methods, providing a more stable and scalable approach for training policies in challenging environments.

5.1 Advantage Functions

As already discussed, REINFORCE is a Monte Carlo method, meaning that it must wait until the end of the episode to obtain a single sample of the return, G_t. G_t being a single sample, it is an unbiased high variance estimate of the true expected return. The high variance comes from the fact that G_t accumulates delayed rewards over potentially long episodes. This makes REINFORCE a slow learning algorithm. We have also seen that variance can be reduced by subtracting a baseline from G_t, such as $\hat{v}(S_t, \mathbf{w})$. In fact, there are many alternative methods to modulate the policy gradient when updating the policy parameters. To cope with this diversity, the update equation can be reformulated in the following general form:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \Phi_t \nabla_{\boldsymbol{\theta}} \ln \pi(A_t \mid S_t, \boldsymbol{\theta}_t),$$

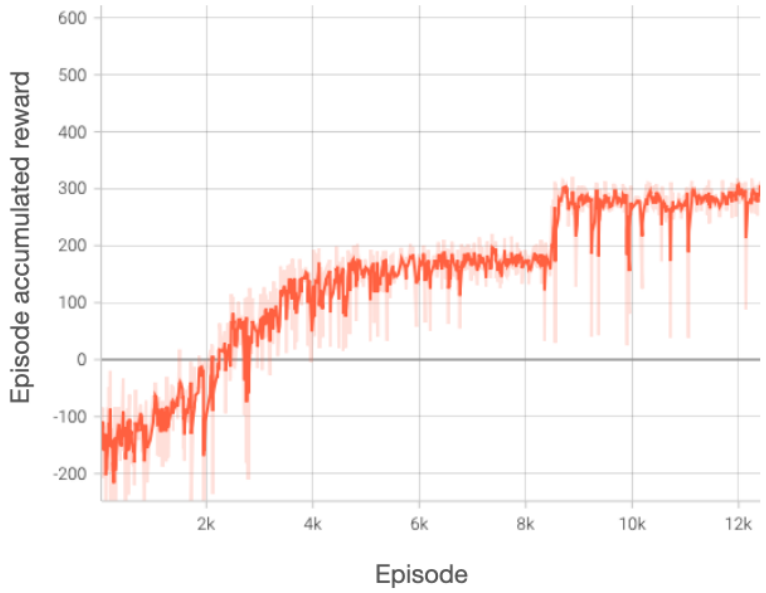


Figure 22: REINFORCE algorithm using MLPs for both the policy and state-value functions in the LunarLander-v2 problem (baseline used). The plot presents the accumulated reward (undiscounted), per learning episode, with unsmoothed data shown in lower opacity.

where Φ_t can be defined in many different ways, including:

$$\Phi_t \doteq \begin{cases} G_t & \text{if in REINFORCE,} \\ G_t - \hat{v}(S_t, \mathbf{w}) & \text{if in REINFORCE with baseline } \hat{v}(S_t, \mathbf{w}), \\ \hat{\Delta}_t & \text{if in Actor-Critic,} \end{cases}$$

where $\hat{\Delta}_t$ is an estimator of what is known as *advantage function*. When using $\Phi_t \doteq \hat{\Delta}_t$, it is common to call the resulting policy gradient algorithm of an *actor-critic* algorithm. The reason why and the definition and purpose of $\hat{\Delta}_t$ are detailed in the following paragraphs.

With $\Phi_t \doteq G_t - \hat{v}(S_t, \mathbf{w})$, the policy function parameters are updated in such a way that the executed action A_t becomes more probable if the sampled return G_t is above the predicted average return (i.e., $G_t - \hat{v}(S_t, \mathbf{w}) > 0$) and less probable if G_t is below the predicted average return (i.e., $G_t - \hat{v}(S_t, \mathbf{w}) < 0$). Hence, $G_t - \hat{v}(S_t, \mathbf{w})$ operates as an estimator of the relative *advantage* of executing action A_t relative to randomly sampling an action according to policy π. However, this advantage estimator still suffers from two problems resulting from depending of G_t: it has high variance and can only be computed by the end of the episode. High variance slows learning and waiting for the end of the episode hampers its application to non-episodic tasks.

We can generalize the idea of *advantage* to define what is known as the *advantage function*, which states how much better it is on average to take a specific action a in state s, over randomly selecting an action according to the policy π. Formally, the advantage function is defined by:

$$\Delta_\pi(s, a) \doteq q_\pi(s, a) - v_\pi(s).$$

From now on, we will index variables with t, which is the time step in policy rollout (i.e., episode) n. Hence, using both indexes allows us to represent data collected in multiple rollouts (i.e., episodes). For instance, the state observed in time step t in rollout n will be represented by $S_{n,t}$.

A well-known estimator of the advantage function is known as *Temporal Difference Error* (TD-Error), which we will denote by δ_t and is defined as follows:

$$\delta_{n,t} \doteq R_{n,t} + \gamma \hat{v}(S_{n,t+1}, \mathbf{w}) - \hat{v}(S_{n,t}, \mathbf{w}).$$

The TD-Error also compares the expected return with the state-value function $\hat{v}(S_{n,t}, \mathbf{w})$. However, instead of estimating the expected return with $G_{n,t}$, the TD-Error estimates it with $R_{n,t} + \gamma \hat{v}(S_{n,t+1}, \mathbf{w})$. It means that the expected return is the reward collected in t plus a discounted prediction of the accumulated discounted reward from $t + 1$ onwards. This alternative estimator of the expected return has

lower variance, thus improving learning. Relying on a prediction that we gradually improve, i.e., the approximate state-value function, rather than in actual observations, is known as *bootstrapping*.

The Monte Carlo estimator $G_{n,t}$ exhibits higher variance because its computation depends on the sum of many future rewards, whose uncertainty compounds over the multiple steps. Conversely, given that the approximate state-value function summarizes the future, its variance is lower. However, this function being an approximation, naturally induces some bias. Hence, we traded-off variance by bias. Furthermore, given that TD-Error only depends on evaluations carried out in t and $t+1$, it can be applied to non-episodic problems.

We can reduce the bias in the advantage function estimator by extending the horizon of observed rewards, instead of considering only the current one. For instance, we can add the reward obtained t, the discounted reward obtained in $t+1$, and only then we add the predicted return from $t+2$ onwards according to $\hat{v}(S_{n,t+2}, \mathbf{w})$. The more we push into the future the evaluation of $\hat{v}$ the more we reduce the bias induced by $\hat{v}$ at the cost of increasing variance due to the higher number of observed reward terms.

Let us generalize the TD-Error advantage estimator for an horizon of $k > 1$ observed reward terms before adding the term $\hat{v}$:

$$
\begin{aligned}
&\hat{\Delta}^{(1)}_{n,t} \doteq \delta_{n,t} = -\hat{v}(S_{n,t}, \mathbf{w}) + R_{n,t} + \gamma \hat{v}(S_{n,t+1}, \mathbf{w}),\\
&\hat{\Delta}^{(2)}_{n,t} \doteq \delta_{n,t} + \gamma\delta_{n,t+1} = -\hat{v}(S_{n,t}, \mathbf{w}) + R_{n,t} + \gamma R_{n,t+1} + \gamma^2 \hat{v}(S_{n,t+2}, \mathbf{w}),\\
&\hat{\Delta}^{(3)}_{n,t} \doteq \delta_{n,t} + \gamma\delta_{n,t+1} + \gamma^2\delta_{n,t+2} = -\hat{v}(S_{n,t}, \mathbf{w}) + R_{n,t} + \gamma R_{n,t+1} + \gamma^2 R_{n,t+2} + \gamma^3 \hat{v}(S_{n,t+3}, \mathbf{w}),\\
&\dots\\
&\hat{\Delta}^{(k)}_{n,t} \doteq \sum_{l=0}^{k-1} \gamma^l \delta_{n,t+l} = -\hat{v}(S_{n,t}, \mathbf{w}) + R_{n,t} + \gamma R_{n,t+1} + \ldots + \gamma^{k-1} R_{n,t+k-1} + \gamma^k \hat{v}(S_{n,t+k}, \mathbf{w}).
\end{aligned}
$$

We can combine these k-step estimators with an exponentially-weighted average to produce what is known as truncated *Generalized Advantage Estimator* (GAE) so as to achieve fine control over the bias-variance trade-off:

$$
\hat{\Delta}_{n,t} \doteq (1-\lambda)\left(\hat{\Delta}^{(1)}_{n,t} + \lambda\hat{\Delta}^{(2)}_{n,t} + \lambda^2\hat{\Delta}^{(3)}_{n,t} + \ldots + \lambda^{k-1}\hat{\Delta}^{(k)}_{n,t}\right) \tag{39}
$$

The truncated *generalized advantage estimator* has two hyper-parameters to control the bias-variance trade-off. The higher the parameter k the higher the control we are able to have over the trade-off, at the cost of computation. The higher the parameter λ the higher variance and the lower the bias. That is the case because we gradually take more into account the actual rewards observed by the agent further in time. Decreasing λ has the inverse effect on variance and bias.

As mentioned, the estimated advantage function $\hat{\Delta}_{n,t}$ is used to guide policy updates by estimating the relative value of taking a particular action in a given state. However, the scale of $\hat{\Delta}_{n,t}$ can vary during training due to several factors: (a) changes in the policy affect the distribution of rewards and value estimates, leading to variations in the computed advantages; (b) the reward structure in the environment (e.g., sparse rewards, dense rewards, or clipped rewards) influences the scale and variability of $\hat{\Delta}_{n,t}$; and uncertainty in the estimated value function $\hat{v}$ introduce noise or inconsistencies in the advantage calculation. These factors can cause the scale of $\hat{\Delta}_{n,t}$ to fluctuate across training iterations, potentially leading to: (a) large advantage values causing overly aggressive updates to the policy; and (b) small advantage values with diminished impact on updates, slowing down convergence.

To ensure a consistent scale and promote more stable and efficient training, the advantage function is normalized. The normalized advantage function, denoted as $\tilde{A}_{n,t}$, is computed as:

$$
\tilde{\Delta}_{n,t} = \frac{\hat{\Delta}_{n,t} - \mu_{\pi_\theta}}{\sigma_{\pi_\theta}}, \tag{40}
$$

where μ_{π_θ} and σ_{π_θ} are the mean and standard deviation of all unnormalized advantage function values $\hat{\Delta}_{n,t}$ collected with the current policy π_θ (more details below).

5.2 Multiple Rollouts

In REINFORCE, training stability and performance can be significantly improved by extending the learning process beyond a single episode or rollout. Although REINFORCE is simple and straightforward,

it computes policy gradients based on a single trajectory. This approach can introduce high variance into the updates, which may slow learning and lead to instability, as the gradient estimation may not accurately represent the overall distribution of states and actions.

To address this, we can use the current policy π_θ to perform N rollouts (i.e., episodes), each consisting of T time steps (ignoring potential early terminations of rollouts for simplicity). During these rollouts, all actions taken, as well as observed states and rewards, are recorded for later use. Collecting experiences from multiple rollouts helps to average out the inherent randomness and provides more reliable gradient estimates. These rollouts can be executed sequentially or in parallel. Parallel execution can greatly accelerate data collection and is commonly utilized in practice to take advantage of multi-threaded or distributed computing environments.

After running the N rollouts, we compute the discounted returns for each time step and the corresponding unnormalized advantages. Each experience tuple, comprising the state, action, reward, discounted return, and unnormalized advantage, is stored in an experience buffer D. The data is tagged with the information about the specific rollout and time step from which it was collected. Algorithm 5 details how multiple rollouts can be used to build the experiences buffer.

Algorithm 5 CreateExperiencesBuffer (π, $\boldsymbol{\theta}$, $\hat{v}$, $\boldsymbol{w}$, N, T, γ, λ)

Input: policy function π and its parameterization $\boldsymbol{\theta}$
Input: state-value function, $\hat{v}$, and its current parameterization, $\boldsymbol{w}$
Input: number of rollouts N
Input: number of time steps per rollout, T
Input: discount factor, γ, and weight factor for advantage, λ

// Perform N rollouts with the agent in the environment
For each rollout $n = 0, 1, \ldots, N-1$
  Observe initial state $S_{n,0}$
  For each time step $t = 0, 1, \ldots, T-1$
    Sample action from policy, $A_{n,t} \sim \pi(\cdot \mid S_{n,t}, \boldsymbol{\theta})$
    Execute action $A_{n,t}$, observe reward $R_{n,t+1}$ and next state $S_{n,t+1}$

// Process the rollouts and add them to the experiences buffer
Reset experiences buffer, $D \leftarrow []$
For each rollout $n = 0, 1, \ldots, N-1$
  For each time step $t = 0, 1, \ldots, T-1$
    Compute discounted return $G_{n,t} \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_{n,k}$
    Compute unnormalized advantage $\hat{\Delta}_{n,t}$ with Equation 39 (with λ, $\hat{v}$, $\boldsymbol{w}$, $\{R_{n,i}\}_{i \geq t}$, $\{S_{n,i}\}_{i \geq t}$)
    Append data to buffer, $D \leftarrow D \cup (S_{n,t}, A_{n,t}, R_{n,t+1}, G_{n,t}, \hat{\Delta}_{n,t})$

Return D

5.3 Mini-Batches

Experiences stored in the experiences buffer D often exhibit correlations due to the way rollouts are generated by the same policy. This correlation can lead to overrepresentation of certain state-action-reward-return-advantage tuples while potentially underrepresenting others that are also important for the optimal policy. Using the entire buffer for a single gradient update can introduce high variance, as dominant segments may skew the gradient estimate, resulting in noisy and unstable learning.

To address this issue, it is beneficial to estimate the gradient using mini-batches of randomly sampled experiences. Each mini-batch should contain uncorrelated experiences across rollouts and time steps, ensuring a more representative and unbiased gradient estimate over the entire buffer.

To create M-length mini-batches (with M as a user-defined hyperparameter), we first shuffle the experiences buffer D to randomize the order of its elements and break existing correlations. The shuffled buffer is denoted by D_{shuffled}. The first M experiences from D_{shuffled}, denoted as $D_{\text{shuffled}}[0 : M-1]$, are selected to form a mini-batch D_{mini}. Before populating D_{mini}, the normalized advantage for each experience tuple in $D_{\text{shuffled}}[0 : M-1]$ is computed. Each experience tuple is then augmented with its normalized advantage and added to D_{mini}. Then, the mini-batch D_{mini} is added to a list of mini-batches B and the elements belonging to D_{mini} are removed from D_{shuffled}. This process repeats until D_{shuffled}

is left out of experiences, $|D_{\text{shuffled}}| = 0$. When the last remaining elements in D_{shuffled} are not enough to fill a last mini-batch, the said mini-batch can be filled with randomly selected experiences from the mini-batches stored in the list B. Other strategies could be selected to handle these last experiences. Algorithm 6 outlines this procedure for constructing the mini-batch list B.

Algorithm 6 CreateMiniBatches (D, M)

Input: experiences buffer, D
Input: mini-batch size, M

Initialize list of mini-batches, $B \leftarrow []$
Shuffle the experience buffer, $D_{\text{shuffled}} \leftarrow \text{Shuffle}(D)$
While $|D_{\text{shuffled}}| > 0$
  If $|D_{\text{shuffled}}| < M$
    Randomly sample $M - |D_{\text{shuffled}}|$ experiences from B, Z
    Append the sampled experiences Z to D_{shuffled}, $D_{\text{shuffled}} \leftarrow D_{\text{shuffled}} \cup Z$
  Compute average of all unnormalized advantages stored in $D_{\text{shuffled}}[0 : M-1]$, μ_{π_θ}
  Compute standard deviation of all unnormalized advantages stored in $D_{\text{shuffled}}[0 : M-1]$, σ_{π_θ}
  Initialize a mini-batch, $D_{\text{mini}} \leftarrow []$
  For each experience $(S_t^{(n)}, A_t^{(n)}, R_{t+1}^{(n)}, G_t^{(n)}, \hat{\Delta}_t^{(n)})$ **in** $D_{\text{shuffled}}[0 : M-1]$
    Compute normalized advantage $\tilde{\Delta}_t \leftarrow (\hat{\Delta}_t - \mu_{\pi_\theta})/(\sigma_{\pi_\theta})$ (Equation 40)
    Add experience to mini-batch, $D_{\text{mini}} \leftarrow D_{\text{mini}} \cup (S_t^{(n)}, A_t^{(n)}, R_{t+1}^{(n)}, G_t^{(n)}, \tilde{\Delta}_t^{(n)})$
  Add mini-batch to list of batches, $B \leftarrow B \cup D_{\text{mini}}$
  Remove the mini-batch from the shuffled buffer, $D_{\text{shuffled}} \leftarrow D_{\text{shuffled}} \setminus D_{\text{shuffled}}[0 : M-1]$

Return B

5.4 Surrogate Objective

Recall that the REINFORCE algorithm with non-linear policies optimizes a policy by maximizing the following objective function, which has been adapted to include the time index t that occurred at rollout n:

$$\mathcal{L}_{n,t}^{\pi}(\boldsymbol{\theta}) \doteq (G_{n,t} - \hat{v}(S_{n,t}, \mathbf{w})) \ln \pi(A_{n,t}|S_{n,t}, \boldsymbol{\theta}),$$

where $G_{n,t}$ is the actual return received after time t in rollout n, $\hat{v}(S_{n,t}, \mathbf{w})$ is the estimated value function (baseline), and $\ln \pi(A_{n,t}|S_{n,t}, \boldsymbol{\theta})$ is the log-probability of taking action $A_{n,t}$ given state $S_{n,t}$ under the current policy parameterized by $\boldsymbol{\theta}$.

The objective function can be maximized by means of stochastic gradient ascent. That is, the parameters of the policy are locally optimized by following the estimated gradient. The gradient can be automatically estimated by means of differentiation toolkits that make use of some form of back-propagation.

$G_{n,t} - \hat{v}(S_{n,t}, \mathbf{w})$ is a single-sample estimate of the advantage of executing $A_{n,t}$. Hence, the intuition behind REINFORCE objective function is that the policy parameters are updated in the direction that maximizes the expected advantage, scaled by the log-probability of the chosen action. If $G_{n,t}$ (the return) is greater than the baseline $\hat{v}(S_{n,t}, \mathbf{w})$ (i.e., A_t is advantageous), the policy is encouraged to increase the probability of selecting $A_{n,t}$; otherwise, it is penalized.

A significant drawback of the REINFORCE algorithm is the high variance in the estimation of the advantage due to the dependence on individual sample returns $G_{n,t}$. This variance can make training unstable and slow, as updates can be erratic. As we have already seen, a way to mitigate this is to use the normalized estimated advantage function $\tilde{\Delta}_{n,t}$ as a more stable estimate of the advantage of performing action A_t, resulting in the following objective function:

$$\mathcal{L}_{n,t}^{+,\pi}(\boldsymbol{\theta}) \doteq \tilde{\Delta}_{n,t} \ln \pi(A_{n,t}|S_{n,t}, \boldsymbol{\theta}).$$

Clipped Objective

As discussed, the learning process happens by performing steps in parameter space in the gradient direction, these steps being scaled by a learning rate α. This learning rate needs to be set small for the

following reason. A step along the gradient ensures that the probability of the selected action grows in case it is associated with high advantage. The problem is that we have little control over how much it grows, as the gradient depends on the shape of the policy function in the neighborhood of its current parameterization. The shape changes as learning progresses, meaning that a large step size may result in a large change in action probabilities distribution, potentially resulting in catastrophic forgetting. Hence, the learning update needs to be small as even minor variations in parameter space can lead to significant changes in performance, where a single poor update might drastically degrade policy effectiveness.

The challenge here is to allow larger steps in parameter space without causing unstable updates. This requires an approach that considers not just the gradient in parameter space but also how much the policy's action distribution changes in response to these updates. The algorithm Proximal Policy Optimization (PPO) introduced a way to address this challenge with a set of new mechanisms.

First, it is important to note that PPO operates on mini-batches. That is to say that it performs several gradient ascent steps in each mini-batch before proceeding to process the next one. On the one hand, it leverages the benefits of mini-batches to stabilize learning, as discussed earlier. On the other hand, it is much more sample efficient than performing a policy rollout per gradient ascent step. Moreover, the PPO objective function is designed to prevent excessively large updates that could destabilize training. As it will be shown, this objective function is not optimizing the true objective, defined by $\mathcal{L}^{+,\pi}_{n,t}(\boldsymbol{\theta})$, and so it is often called "surrogate" objective function. The definition of this objective is detailed next.

Let us denote $\boldsymbol{\theta}$ as the policy's parameter vector that is being updated in the current mini-batch and $\boldsymbol{\theta}_{\text{old}}$ policy's parameter vector at the last update in the previous mini-batch. Hence, when we terminate processing a mini-batch we freeze the current policy parameter vector, $\boldsymbol{\theta}_{\text{old}} \leftarrow \boldsymbol{\theta}$. Bearing this in mind, the PPO "surrogate" objective function is:

$$\mathcal{L}^{\text{CLIP},\pi}_{n,t}(\boldsymbol{\theta};\epsilon_\pi) \doteq \min\left(r_{n,t}(\boldsymbol{\theta})\tilde{\Delta}_{n,t}, \text{clip}\left(r_{n,t}(\boldsymbol{\theta}), 1-\epsilon_\pi, 1+\epsilon_\pi\right)\tilde{\Delta}_{n,t}\right),$$

with:

$$r_{n,t}(\boldsymbol{\theta}) \doteq \frac{\pi(A_{n,t}|S_{n,t},\boldsymbol{\theta})}{\pi(A_{n,t}|S_{n,t},\boldsymbol{\theta}_{\text{old}})}$$

where ϵ_π is a small positive constant defined as an hyper-parameter, and $\text{clip}(x,a,b) \doteq \min(\max(x,a),b)$.

The probabilities ratio $r_{n,t}(\boldsymbol{\theta})$ represents how much the probability of the selected action has changed between the new policy $\pi(\boldsymbol{\theta})$ and the old policy $\pi(\boldsymbol{\theta}_{\text{old}})$. In a sense, as in REINFORCE, by maximizing $r_{n,t}(\boldsymbol{\theta})\tilde{\Delta}_{n,t} = \frac{\pi(A_{n,t}|S_{n,t},\boldsymbol{\theta})}{\pi(A_{n,t}|S_{n,t},\boldsymbol{\theta}_{\text{old}})}\tilde{\Delta}_{n,t}$ we are maximizing $\pi(A_{n,t}|S_{n,t},\boldsymbol{\theta})\tilde{\Delta}_{n,t}$, the difference being that the gradient is being scaled by $\pi(A_{n,t}|S_{n,t},\boldsymbol{\theta}_{\text{old}})$ (recall that $\boldsymbol{\theta}_{\text{old}}$ is fixed while processing the current mini-batch).

The term $\text{clip}(r_{n,t}(\boldsymbol{\theta}), 1-\epsilon_\pi, 1+\epsilon_\pi)\hat{A}_{n,t}$ clips the probability ratio $r_{n,t}(\boldsymbol{\theta})$ to remove the incentive for the maximization process to push $r_{n,t}(\boldsymbol{\theta})$ outside the interval $[1-\epsilon_\pi, 1+\epsilon_\pi]$, thus limiting the extent to which the policy's probability ratio can change. This clipping restricts how the new probability for action $A_{n,t}$ can deviate from the probability assigned in the previous mini-batch, ensuring that updates to the policy remain stable.

By taking the minimum of the clipped and unclipped terms, the final objective becomes a lower (pessimistic) bound on the unclipped objective. This formulation ensures that changes in the probability ratio induced by the clipping procedure that would lead to an excessive improvement in the objective are ignored, while changes that would worsen the objective are accepted. Consequently, this pessimistic approach helps reduce gradient magnitudes and, in turn, lowers the variance in gradient estimation. These safeguards enable PPO to take larger steps in parameter space, promoting faster learning without causing instability in action probabilities.

KL Penalty

The PPO objective function can be formalized differently. Instead of employing a clipping procedure in the probability ratio, the alternative formulation imposes a penalty on those policy parameterizations $\boldsymbol{\theta}$ that explicitly induce an action probability distribution too dissimilar from the one induced by $\boldsymbol{\theta}_{\text{old}}$. The dissimilarity between the distributions is computed using the Kullback-Leibler (KL) divergence (see Section 1.13). Intuitively, the KL divergence measures how much the new policy diverges from the old one. A small KL divergence indicates similar policies, while a large value signals significant changes in action probabilities.

Formally, given a discrete action space $\mathcal{A}$, the KL divergence between the policies induced by the updated parameters $\boldsymbol{\theta}$ and the old parameters $\boldsymbol{\theta}_{\text{old}}$ quantifies the expected difference in the information

content (or surprise) between the two policies. It is computed as the expected logarithmic ratio of the probabilities assigned to actions by the updated policy $\pi_{\boldsymbol{\theta}}$ and the old policy $\pi_{\boldsymbol{\theta}_{\text{old}}}$ (by applying Equation 5 from Section 1.13):

$$D_{n,t}^{\text{KL},\pi}(\boldsymbol{\theta}, \boldsymbol{\theta}_{\text{old}}) \doteq \mathbb{E}_{a\sim\pi_{\boldsymbol{\theta}}}\left[\ln \frac{\pi(a|S_{n,t}, \boldsymbol{\theta})}{\pi(a|S_{n,t}, \boldsymbol{\theta}_{\text{old}})}\right] = \sum_{a\in\mathcal{A}} \pi(a|S_{n,t}, \boldsymbol{\theta}) \ln \frac{\pi(a|S_{n,t}, \boldsymbol{\theta})}{\pi(a|S_{n,t}, \boldsymbol{\theta}_{\text{old}})}. \tag{41}$$

The objective function based with KL penalty is defined as follows, where β is a positive constant hyper-parameter that determines the weight of the penalty within the overall objective:

$$\mathcal{L}_{n,t}^{\text{KL},\pi}(\boldsymbol{\theta}; \beta) \doteq r_{n,t}(\boldsymbol{\theta})\tilde{\Delta}_{n,t} - \beta D_{n,t}^{\text{KL},\pi}(\boldsymbol{\theta}, \boldsymbol{\theta}_{\text{old}}).$$

Entropy Regularization

To further improve the stability of the training and promote exploration, an *entropy* term can be added to the PPO objective. In general, entropy of a random variable quantifies the average level of uncertainty associated with the variable's possible outcomes (see Section 1.12). The entropy of the policy π parameterized by $\boldsymbol{\theta}$ and conditioned on state $S_{n,t}$, $\mathcal{H}_{n,t}^{\pi}(\boldsymbol{\theta})$, is defined as (by applying Equation 2 from Section 1.12):

$$\mathcal{H}_{n,t}^{\pi}(\boldsymbol{\theta}) \doteq -\sum_{a\in\mathcal{A}} \pi(a|S_{n,t}, \boldsymbol{\theta}) \ln \pi(a|S_{n,t}, \boldsymbol{\theta}).$$

The entropy of the policy quantifies the randomness or uncertainty in the action selection process. It measures how spread out the probability distribution over actions is. A more uniform distribution corresponds to higher entropy, reflecting greater randomness and uncertainty about which action will be selected. Conversely, low entropy indicates that probability is concentrated around one or a few actions, resulting in a more deterministic behavior.

To encourage exploration, the entropy term is added to the PPO objective function as a regularizer. This term penalizes low-entropy policies, as these are less likely to explore alternative actions. By maintaining higher entropy, the policy remains more exploratory, avoiding overconfidence in specific actions and reducing the risk of converging to suboptimal solutions. The entropy term promotes a balance between exploitation (selecting actions with high expected return) and exploration (trying new actions to discover better strategies).

Aggregate Objective

We can combine the clip-based objective, $\mathcal{L}_{n,t}^{\text{CLIP},\pi}(\boldsymbol{\theta}; \epsilon)$, with the KL-based objective, $\mathcal{L}_{n,t}^{\text{KL},\pi}(\boldsymbol{\theta}; \beta)$, and the entropy regularization, $\mathcal{H}_{n,t}^{\pi}(\boldsymbol{\theta})$, in a single objective as follows:

$$\mathcal{L}_{n,t}^{\text{PPO},\pi}(\boldsymbol{\theta}; \nu, \epsilon_\pi, \beta, \eta) \doteq \nu\mathcal{L}_{n,t}^{\text{CLIP},\pi}(\boldsymbol{\theta}; \epsilon_\pi) + (1-\nu)\mathcal{L}_{n,t}^{\text{KL},\pi}(\boldsymbol{\theta}; \beta) + \eta\mathcal{H}_{n,t}^{\pi}(\boldsymbol{\theta}), \tag{42}$$

where $\nu \in \{0, 1\}$ is a binary hyper-parameter used to select between the clip-based and the KL-based sub-objectives and η determines weight of the entropy term in the objective function, i.e., the level of exploration.

5.5 Value Objective

Recall that the loss $\mathcal{L}_{n,t}^{\text{PPO},\pi}(\boldsymbol{\theta}; \nu, \epsilon_\pi, \beta, \eta)$ (Equation 42) depends on the normalized estimated advantage function $\tilde{\Delta}_{n,t}$ (Equation 40), which itself depends on the estimated state-value function $\hat{v}(S_{n,t}, \mathbf{w})$. Therefore, it is necessary to update the value function parameters $\mathbf{w}$.

In REINFORCE, the parameters $\mathbf{w}$ of $\hat{v}(S_{n,t}, \mathbf{w})$ are updated according to the loss $(G_t - \hat{v}(S_t, \mathbf{w}))^2$ (Equation 37). Similarly, in PPO, the value function is learned using the following loss function:

$$\mathcal{L}_{n,t}^{\text{PPO},\hat{v}}(\mathbf{w}) \doteq \left(G_{n,t} - \hat{v}(S_{n,t}, \mathbf{w})\right)^2. \tag{43}$$

5.6 The PPO Algorithm

As explained, the Proximal Policy Optimization (PPO) algorithm optimizes a policy by iteratively improving its parameters to maximize a surrogate objective function while maintaining exploration through entropy regularization.

PPO begins with the initialization of the policy and value function networks' parameters, $\boldsymbol{\theta}$ and $\boldsymbol{w}$. The technique known as *orthogonal initialization* is often used for this purpose, ensuring that the initial weights are uncorrelated and have unit variance. Compared to standard random initialization, orthogonal initialization can improve the stability and efficiency of training by providing a more balanced starting point for the gradient updates, especially in deep networks with many layers.

The training process is repeated over I (a user-defined constant) iterations, which can be early stopped as in REINFORCE. In each iteration, Algorithm 5 is executed to collect an experiences buffer D filled with (state, action, reward, discounted return, unnormalized advantage estimate) tuples observed over N policy rollouts with horizon T.

After filling D, PPO stores the current policy parameters in $\boldsymbol{\theta}_{\text{old}}$ for later use, $\boldsymbol{\theta}_{\text{old}} \leftarrow \boldsymbol{\theta}$. Thus, $\boldsymbol{\theta}_{\text{old}}$ refers to the policy parameters at the onset of the iteration. The policy $\pi_{\boldsymbol{\theta}_{\text{old}}}$ operates as a reference against which $\pi_{\boldsymbol{\theta}}$ is compared as learning proceeds during the current iteration.

PPO then performs E (a user-defined constant) learning epochs to update $\pi_{\boldsymbol{\theta}}$ before moving to the next iteration. Each learning epoch begins by shuffling D and splitting it into M-length mini-batches, organized into a list B. For each mini-batch $D_{\text{mini}} \in B$, the following steps are performed: (1) The gradient of the average policy loss (Equation 42) over the mini-batch is computed; (2) The gradient of the average state-value loss over the mini-batch is computed; (3) The gradients computed in steps 1 and 2 are used to update the parameters of the policy ($\boldsymbol{\theta}$) and state-value ($\boldsymbol{w}$) networks, respectively.

With multiple gradient steps, PPO converges faster, but this also increases the risk of the policy drifting excessively, potentially causing instability. To assess whether the policy has changed too much during this process, the action probability distribution induced by π under $\boldsymbol{\theta}$ can be compared to the one induced by $\boldsymbol{\theta}_{\text{old}}$. Concretely, if the average KL divergence between the action distributions induced by both $\boldsymbol{\theta}$ and $\boldsymbol{\theta}_{\text{old}}$ exceeds a user-defined threshold ξ, the current iteration is stopped early, prompting the agent to perform new rollouts in the environment to gather fresh data.

Algorithm 7 outlines the pseudo-code of the original PPO extended with early stopping based on KL divergence. PPO can be extended in many directions to address the challenges of specific problems effectively. Some possible extensions include:

- Changing the loss function to discourage large policy changes by including a KL penalty instead of applying clipping to the probability ratio
- Implementing a learning rate scheduler (e.g., exponential decay), as we have done in REINFORCE, can help stabilize training as learning progresses, ensuring consistent improvements without overshooting or stagnation.
- Similar to policy updates, applying clipping to value function updates can prevent overfitting or divergence, particularly in environments with high reward variance.
- Using a decaying entropy coefficient helps reduce exploration over time, similarly to a learning rate scheduler, allowing the agent to focus on exploiting learned policies as it gains confidence.
- Normalizing input states, as we have done in REINFORCE, especially in high-dimensional environments, can lead to faster convergence and more stable training.
- Distributing the processing with multiple workers to scale for solving complex problems with extensive state and action spaces.

Note that PPO is a complex algorithm that can be adjusted in various ways to address the diversity of RL problems. Stable Baselines 3 (SB3) includes reliable PyTorch implementations of RL algorithms, including PPO. Consulting its PPO source files is an excellent way to complement this reading.

5.7 Continuous Action Spaces

In the previous section we discussed PPO in the context of discrete actions sampled from a stochastic policy. As discussed, sampling an action from a stochastic policy function involves learning the probability distribution over a set of possible actions. This approach is feasible when the action space is discrete,

Algorithm 7 PPO (α_π, $\alpha_{\mathbf{w}}$, I, E, N, T, M, π, $\hat{v}$, ξ, ν, ϵ_π, β, η)

Input: policy learning rate $\alpha_\pi > 0$ and state-value learning rate $\alpha_{\mathbf{w}} > 0$
Input: number of learning epoches E
Input: number of iterations I
Input: number of rollouts N, number of time steps per rollout T, and mini-batch size M
Input: policy function, π, and state-value function $\hat{v}$
Input: KL divergence threshold $\xi > 0$
Input: Selector ν between $\mathcal{L}^{\text{CLIP}}$ and $\mathcal{L}^{\text{KL}}$
Input: Policy ratio clip range ϵ_π
Input: KL penalty weight β
Input: Entropy regularization weight η

Initialize network parameters with orthogonal initialization
$\boldsymbol{\theta} \leftarrow$ OrthogonalInitialization(), $\mathbf{w} \leftarrow$ OrthogonalInitialization()
Train the networks for I iterations
For each iteration $i = 1, 2, \ldots, I$:
  Create a buffer of experiences using the current policy and its parameters.
  $D \leftarrow$ CreateExperiencesBuffer$(\pi, \boldsymbol{\theta}, \hat{v}, \mathbf{w}, N, T, \gamma, \lambda)$ [use Algorithm 5]
  Reset control flag for learning epoch early stop
  stop $\leftarrow$ false
  Store the current policy parameters.
  $\boldsymbol{\theta}_{\text{old}} \leftarrow \boldsymbol{\theta}$
  Iterate over each learning epoch
  For each learning epoch $e = 1, 2, \ldots, E$:
    Generate mini-batches from the experiences buffer.
    $B \leftarrow$ CreateMiniBatches(D, M)) [use Algorithm 6]
    Iterate over each mini-batch
    For each mini-batch $D_{\text{mini}} \in B$:
      Calculate the gradient for the policy update using the PPO objective.
      $g_\pi \leftarrow \nabla_\theta \left(\frac{1}{M} \sum_{\forall (S_{n,t}, A_{n,t}, R_{n,t+1}, G_{n,t}, \tilde{\Delta}_{n,t}) \in D_{\text{mini}}} \mathcal{L}^{\text{PPO},\pi}_{n,t}(\boldsymbol{\theta}; \nu, \epsilon_\pi, \beta, \eta) \right)$
      Compute the gradient for the value function update using mean squared error.
      $g_{\boldsymbol{w}} \leftarrow \nabla_{\mathbf{w}} \left(\frac{1}{M} \sum_{\forall (S_{n,t}, A_{n,t}, R_{n,t+1}, G_{n,t}, \tilde{\Delta}_t) \in D_{\text{mini}}} \mathcal{L}^{\text{PPO},\hat{v}}_{n,t}(\mathbf{w}) \right)$
      Update the policy parameters using the policy gradient.
      $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_\pi \cdot g_\pi$
      Update the value function parameters using the gradient of value function.
      $\mathbf{w} \leftarrow \mathbf{w} - \alpha_{\mathbf{w}} \cdot g_{\mathbf{w}}$
      If average KL divergence is too excessive early stop learning epoch.
      If $\frac{1}{M} \sum_{\forall (S_{n,t}, A_{n,t}, R_{n,t+1}, G_{n,t}, \tilde{\Delta}_{n,t}) \in D_{\text{mini}}} D^{\text{KL},\pi}_{n,t}(\boldsymbol{\theta}, \boldsymbol{\theta}_{\text{old}}) > \xi$
        stop $\leftarrow$ true
        break [leave for each mini-batch loop]
    If *stop*:
      break [leave for each learning epoch loop]

Return the final policy parameters.
Return $\boldsymbol{\theta}$

as there are a finite number of possible actions, and one can assign probabilities to each. However, in continuous control problems, where the action space consists of real-valued actions (e.g., the torque to be applied to a joint in a robotic arm), the situation is more challenging. In such cases, learning the probability of each individual action would imply learning the probabilities of an infinite number of actions, as the action space consists of all real numbers within a given domain.

An alternative to using a stochastic policy is to use a deterministic policy function. A deterministic policy maps the current state directly to a single action, which simplifies the decision-making process. Specifically, the deterministic policy function outputs a specific action given the current state, and this action is then executed. This approach has the advantage that the policy function needs to be evaluated only once per time step, as there is no need to sample from a distribution over actions.

However, one major drawback of deterministic policies is that they do not inherently support exploration during training. Since a deterministic policy always outputs the same action for a given state, it lacks the variability needed to explore different parts of the action space. Exploration is crucial in reinforcement learning, especially during the early stages of training, to ensure that the policy does not prematurely converge to suboptimal actions. To address this, one must explicitly manage the exploration-exploitation trade-off when using deterministic policies. A common method to achieve this is the ϵ-greedy strategy, in which the agent takes a random action with probability ϵ at each time step, and with probability $1 - \epsilon$, it follows the deterministic policy. This way, the agent balances between exploration (random actions) and exploitation (following the policy's recommendations).

On the other hand, stochastic policies naturally handle this trade-off. In a stochastic policy, actions are sampled from a probability distribution, which allows for exploration without needing external mechanisms. Instead of deterministically choosing an action, the policy outputs a distribution over possible actions, and actions are sampled according to their probabilities. This probabilistic sampling introduces variability in the agent's behavior, ensuring that it explores different actions even when the same state is encountered multiple times.

While stochastic policies excel at exploration, managing continuous action spaces introduces additional complexities. In continuous control problems, explicitly assigning probabilities to every possible action is infeasible due to the infinite size of the action space. Instead, a practical approach is to model the policy as a parametric probability distribution conditioned on the state.

Gaussian Policies

A common approach for continuous action spaces is to learn the parameters of a probability distribution over actions, such as the Gaussian distribution, as presented in Section 1.3. The idea is to model the distribution's statistics, its mean and standard deviation, as functions of the state. The mean represents the most likely action for the current state, while the standard deviation determines the spread of probabilities around the mean, effectively encoding the exploration space. This learned distribution is then sampled to produce the current action, enabling the policy to handle an infinite set of possible actions. In contrast, the discrete case directly learns the probability of each possible action.

To employ a Gaussian distribution as a stochastic policy, both the mean and standard deviation are parametrized as functions of the state, modeled using MLPs. These functions are denoted as $\mu(S_{n,t}, \boldsymbol{\theta})$ and $\sigma(S_{n,t}, \boldsymbol{\theta})$, where $S_{n,t}$ is the state at time step t of rollout n and $\boldsymbol{\theta}$ represents the learnable parameters of the policy network. This can be implemented using either two separate MLPs, each producing a single output (one for μ and one for σ), or a single MLP with two outputs, where one output represents the mean and the other the standard deviation. The choice of architecture depends on the complexity of the problem and computational considerations. This leads to the following formulation of the stochastic policy, modeled as a Gaussian PDF:

$$\pi(A_{n,t} \mid S_{n,t}, \boldsymbol{\theta}) = \frac{1}{\sigma(S_{n,t}, \boldsymbol{\theta})\sqrt{2\pi}} \exp\left(-\frac{(A_{n,t} - \mu(S_{n,t}, \boldsymbol{\theta}))^2}{2\sigma(S_{n,t}, \boldsymbol{\theta})^2}\right). \tag{44}$$

The policy $\pi(A_{n,t} \mid S_{n,t}, \boldsymbol{\theta})$ is a composite function, consisting of the mean $\mu(S_{n,t}, \boldsymbol{\theta})$ and standard deviation $\sigma(S_{n,t}, \boldsymbol{\theta})$. This compositional structure does not hinder parameter optimization. The gradient of the policy function with respect to the parameters $\boldsymbol{\theta}$ propagates through both $\mu(S_{n,t}, \boldsymbol{\theta})$ and $\sigma(S_{n,t}, \boldsymbol{\theta})$. Thus, during the policy gradient update, the optimization process effectively adjusts both the mean and standard deviation according to the PPO loss function.

Sampling Actions

In PPO with continuous action spaces, actions are sampled from a Gaussian distribution whose parameters depend on the current state. Specifically, at each time step t during rollout n, the action $A_{n,t}$ is drawn from a normal distribution with state-dependent mean and standard deviation:

$$A_{n,t} \sim \mathcal{N}(\mu(S_{n,t}, \boldsymbol{\theta}), \sigma(S_{n,t}, \boldsymbol{\theta})^2).$$

In many environments, the action space is bounded, typically within an interval such as $[-1, 1]$. To ensure the sampled actions respect these bounds, a squashing function, commonly the hyperbolic tangent, is applied to the output of the Gaussian sample:

$$A_{n,t} = \tanh(\tilde{A}_{n,t}), \quad \tilde{A}_{n,t} \sim \mathcal{N}(\mu(S_{n,t}, \boldsymbol{\theta}), \sigma(S_{n,t}, \boldsymbol{\theta})^2).$$

This transformation ensures that the final action remains within the allowed range, while preserving the stochasticity and flexibility of the Gaussian policy. When using such squashing functions, care must be taken when computing action probabilities, as the transformation affects the resulting probability density.

Multidimensional Action Spaces

In multidimensional action spaces, such as when an agent has multiple actuators, the policy must produce actions for each dimension. A common approach is to use multiple Gaussian distributions, where each Gaussian models a single dimension of the action space. The parameters of these Gaussians, namely the means and standard deviations, can be implemented using different architectural designs depending on the problem's complexity and computational requirements.

One approach is to use separate MLPs for each Gaussian, with each MLP independently producing the mean and standard deviation for a single dimension. This setup offers maximum flexibility for learning independent parameterizations across dimensions. Alternatively, a single MLP with multiple outputs can produce the parameters for all Gaussians. In this case, the MLP outputs the means and standard deviations for all dimensions, making it a more compact and computationally efficient solution. A third, hybrid approach, involves using an MLP with shared layers to extract common features across all dimensions of the action space, followed by specialized output layers (or heads) that produce the mean and standard deviation for each Gaussian. This design balances parameter sharing with task-specific flexibility.

The choice of architecture depends on the degree of correlation between dimensions, the complexity of the action space, and the computational resources available. When there are dependencies or shared features across dimensions, using shared layers or a single MLP often works well. In contrast, separate MLPs may be more suitable for cases where dimensions are highly independent.

KL divergence

If the action space is multidimensional with d dimensions, and assuming the Gaussian distributions are factorized across dimensions (i.e., the dimensions are independent), the KL divergence between the old policy $\boldsymbol{\theta}_{\text{old}}$ and the current policy $\boldsymbol{\theta}$ for the i-th dimension is given by Equation 7:

$$D_{n,t}^{\text{KL},\pi,i}(\boldsymbol{\theta}, \boldsymbol{\theta}_{\text{old}}) = \ln\left(\frac{\sigma_i(S_{n,t}, \boldsymbol{\theta}_{\text{old}})}{\sigma_i(S_{n,t}, \boldsymbol{\theta})}\right) + \frac{\sigma_i(S_{n,t}, \boldsymbol{\theta})^2 + (\mu_i(S_{n,t}, \boldsymbol{\theta}) - \mu_i(S_{n,t}, \boldsymbol{\theta}_{\text{old}}))^2}{2\sigma_i(S_{n,t}, \boldsymbol{\theta}_{\text{old}})^2} - \frac{1}{2},$$

where $\mu_i(S_{n,t}, \boldsymbol{\theta})$ and $\sigma_i(S_{n,t}, \boldsymbol{\theta})$ are the mean and standard deviation of the current policy for the i-th action dimension and $\mu_i(S_{n,t}, \boldsymbol{\theta}_{\text{old}})$ and $\sigma_i(S_{n,t}, \boldsymbol{\theta}_{\text{old}})$ are the mean and standard deviation of the old policy for the i-th action dimension.

The total KL divergence over the d dimensions, assuming independence between dimensions, is the sum of the KL divergences for each dimension:

$$D_{n,t}^{\text{KL},\pi}(\boldsymbol{\theta}, \boldsymbol{\theta}_{\text{old}}) = \sum_{i=1}^{d} D_{n,t}^{\text{KL},\pi,i}(\boldsymbol{\theta}, \boldsymbol{\theta}_{\text{old}}).$$

Information Entropy

If the action space is multidimensional with d dimensions, and assuming the Gaussian distributions are factorized across dimensions (i.e., the dimensions are independent), the entropy of the current policy $\boldsymbol{\theta}$ for the i-th dimension is given by Equation 4:

$$\mathcal{H}_{n,t}^{\pi,i}(\boldsymbol{\theta}) = \frac{1}{2} + \frac{1}{2}\ln(2\pi) + \ln\sigma_i(S_{n,t}, \boldsymbol{\theta}),$$

where $\sigma_i(S_{n,t}, \boldsymbol{\theta})$ is the standard deviation of the current policy for the i-th action dimension.

The total information entropy over the d dimensions, assuming independence between dimensions, is the sum of the information entropy for each dimension:

$$\mathcal{H}_{n,t}^{\pi}(\boldsymbol{\theta}) = \sum_{i=1}^{d} \mathcal{H}_{n,t}^{\pi,i}(\boldsymbol{\theta}).$$

5.8 PPO in Action: Examples and Results

The authors of PPO compared it against REINFORCE, also known as Vanilla Policy Gradient (VPG), across various tasks, including continuous control problems in the physics-based simulator MuJoCo[3]. Figure 23 shows snapshots of four representative MuJoCo environments: HalfCheetah, Hopper, Reacher, and Swimmer.

The environments showcase diverse robot morphologies and objectives:

- **HalfCheetah**[4]: A 2D bipedal robot whose goal is to maximize its forward velocity.
- **Hopper**[5]: A 2D monopod robot designed to hop forward as quickly as possible.
- **Swimmer**[6]: A 2D snake-like robot tasked with propelling itself forward in a simulated fluid environment.
- **Reacher**[7]: A 2D jointed robotic arm with the objective of moving its end-effector as close as possible to a randomly placed target.

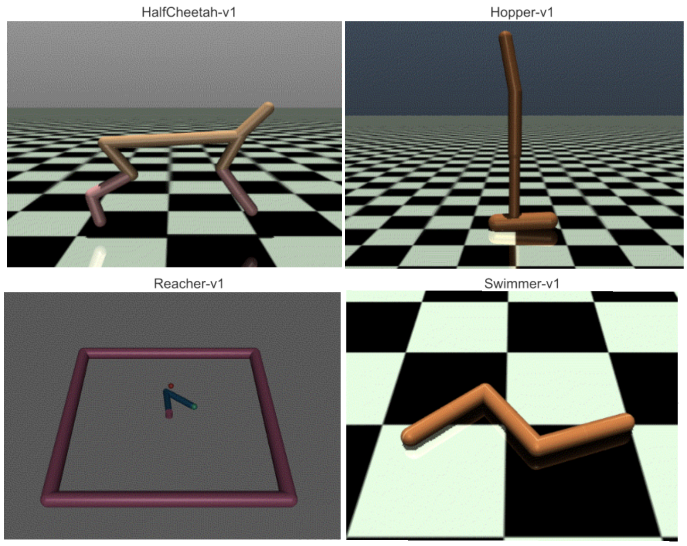


Figure 23: Snapshots of four environments in MuJoCo: HalfCheetah, Hopper, Reacher, and Swimmer.

[3] https://mujoco.org
[4] https://www.gymlibrary.dev/environments/mujoco/half_cheetah/
[5] https://www.gymlibrary.dev/environments/mujoco/hopper/
[6] https://www.gymlibrary.dev/environments/mujoco/swimmer/
[7] https://www.gymlibrary.dev/environments/mujoco/reacher/

Figure 24 illustrates the learning performance of PPO and VPG across these environments, as reported in [7]. The learning curves clearly demonstrate that PPO consistently outperforms VPG, highlighting its robustness and efficiency in optimizing policies for complex tasks. For clarity, additional comparisons to other methods have been omitted.

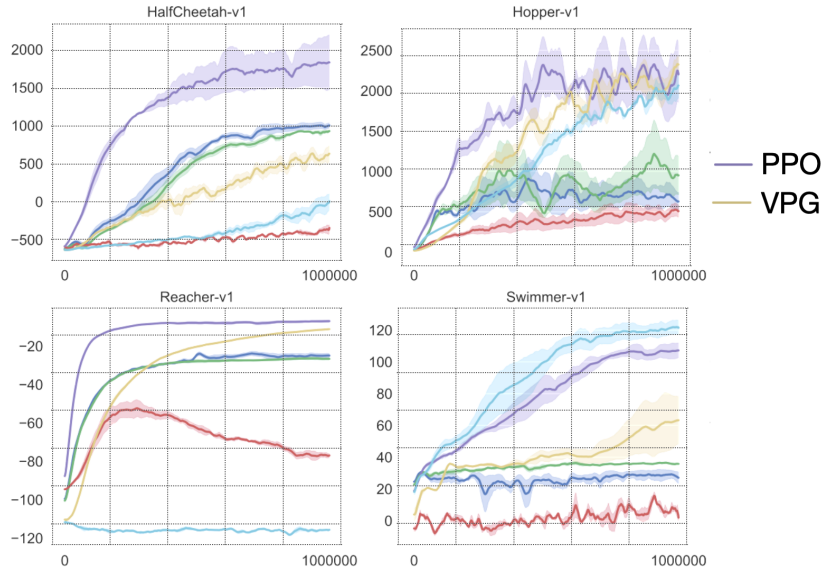


Figure 24: Learning curves of PPO and VPG in four MuJoCo environments (adapted from [7]). Curves for additional methods have been excluded for simplicity.

References

[1] Goodfellow, I., Bengio, Y., & Courville, A. (2016). **Deep Learning**. MIT Press. http://deeplearningbook.org/

[2] Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018, July). **Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor**. In Proceedings of the International conference on machine learning (pp. 1861-1870). PMLR. https://arxiv.org/pdf/1801.01290

[3] Lillicrap, T. P. (2015). **Continuous control with deep reinforcement learning**. arXiv preprint arXiv:1509.02971. https://arxiv.org/pdf/1509.02971

[4] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). **Human-level control through deep reinforcement learning**. nature, 518(7540), 529-533. https://www.nature.com/articles/nature14236

[5] OpenAI (2018). **OpenAI Spinning Up in Deep RL**. https://spinningup.openai.com/

[6] SB3 Team (2024). **Stable Baselines 3**. https://github.com/DLR-RM/stable-baselines3/

[7] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). **Proximal Policy Optimization Algorithms**. https://arxiv.org/pdf/1707.06347

[8] Sutton, R. S., & Barto, A. G. (2018). **Reinforcement Learning: An Introduction**. MIT Press. https://www.andrew.cmu.edu/course/10-703/textbook/BartoSutton.pdf

[9] Williams, R. J. (1992). **Simple statistical gradient-following algorithms for connectionist reinforcement learning**. Machine learning, 8, 229-256.