BDMA - Machine Learning

Jose Antonio Lorencio Abril

Fall 2023

Professor: Tom Dupuis

Student e-mail: jose-antonio.lorencio-abril@student-cs.fr

This is a summary of the course *Machine Learning* taught at the Université Paris Saclay - CentraleSupélec by Professor Tom Dupuis in the academic year 23/24. Most of the content of this document is adapted from the course notes by Dupuis, [1], so I won't be citing it all the time. Other references will be provided when used.

Contents

Part I

Deep Learning

1 Introduction

Artificial Intelligence is a wide concept, encompassing different aspects and fields. We can understand the term AI as the multidisciplinary field of study that aims at recreating human intelligence using artificial means. This is a bit abstract, and, in fact, there is no single definition for what this means. Intelligence is not fully understood, and thus it is hard to assess whether an artificial invention has achieved intelligence, further than intuitively thinking so.

For instance, AI involves a whole variety of fields:

- Perception
- Knowledge
- Cognitive System
- Planning
- Robotics
- Machine Learning (Neural Networks)
- Natural Language Processing

Leveraging all of these, people try to recreate or even surpass human performance in different tasks. For example, a computer program that can play chess better than any human could ever possibly play, such as Stockfish, or a system that is able to understand our messages and reply, based on the knowledge that it has learnt in the past, such as ChatGPT and similar tools. Other examples are self-driving cars, auto-controlled robots, etc.

Therefore, AI is a very wide term, which merges many different scientific fields. **Machine Learning**, on the other side, is a narrower term, which deals with the study of the techniques that we can use to make a computer learn to perform some task. It takes concepts from Statistics, Optimization Theory, Computer Science, Algorithms, etc. A relevant subclass of Machine Learning, which has come to be one of the most prominent fields of research in the recent years, is **Neural Networks** or **Deep Learning**, which consists on an ML technique based on the human brain. Many amazing use cases that we see everywhere, like Siri (Apple assistant), Cortana (Windows assistant), Amazon recommender system, Dall-E (OpenAI image generation system), etc. Not only this, but the trend is growing, and the interest in DL is continuously increasing.

This is partly also due to the increase in computing resources, and the continuous optimization that different techniques are constantly experiencing. For instance, for a model trained on one trillion data points, in 2021 the training process required around 16500x less compute than a model trained in 2012.

But not everything is sweet and roses when using DL. Since these systems are being involved in decision making processes, there are some questions that arise, like whose responsibility is it when a model fails? Moreover, data is needed to train the models, so it is relevant to address how datasets should be collected, and to respect the privacy of the people that produce data. In addition, the recent technologies that are able to generate new content and to modify real content, make it a new issue that AI can create false information, mistrust, and even violence or paranoia.

Nonetheless, let's not focus on the negative, there are lots of nice applications of DL, and it is a key component to deal with data, achieving higher performance than traditional ML techniques for huge amount of data.

1.1 AI History

In 1950, Alan turing aimed to answer the question '*Can machines think?*' through a test, which came to be named the **Turing Test**, and consists in a 3 players game. First, a similar game is the following: 2 talkers, a man and a female, and 1 interrogator. The interrogator asks questions to the talkers, with the aim of determining who is the man and who is the female. The man tries to trick the interrogator, while the woman tries to help him to identify her.

Then, the Turing Test consists in replacing the man by an artificial machine. Turing thought that a machine that could trick a human interrogator, should be considered intelligent.

Later, in 1956, in the Dartmouth Workshop organized by IBM, the term **Artificial Intelligence** was first used to describe *every aspect of learning or any other feature of intelligence can be so precisely described that a machine can be made to simulate it.*

From this year on, there was a focus on researching about **Symbolic AI**, specially in three areas of research:

- Reasoning as search: a different set of actions leads to a certain goal, so we can try to find the best choice of action to obtain the best possible outcome.
- Natural Language: different tools were developed, following grammar and language rules.
- Micro world: small block based worlds, that the system can identify and move.

In 1958, the **Perceptron** was conceived, giving birth to what is called the connectionism, an approach to AI based on the human brain, and a big hype that encouraged funding to support AI research. At this era, scientists experience a bit of lack of perspective, thinking that the power of AI was much higher than it was. For instance, H. A. Simon stated in 1965 that '*machines will be capable, within twenty years, of doing any work a man can do.*' We can relate to our time, with the huge hype that AI is experiencing, as well as the many apocaliptic theories that some people are making. Maybe we are again overestimating the power of AI.

The time from 1974 to 1980 is seen as the first winter of AI, in which research was slowed down and funding was reduced. This was due to several problems found at the time:

- There were few computational resources.
- The models at the time were not scalable.
- The Moravec's paradox: it is comparatively easy to make computers exhibit adult level performance on intelligence test or playing checkers, and difficult or impossible to give them the skills of a one-year-old when it comes to perception and mobility.
- Marvin Minsky made some devastating critics to connectionism, compared to symbolic, rule-based models:
  - Limited capacity: Minsky showed that single-layer perceptrons (a simple kind of neural network) could not solve certain classes of problems, like the XOR problem. While it was later shown that multi-layer perceptrons could solve these problems, Minsky's work resulted in a shift away from neural networks for a time.
  - Lack of clear symbols: Minsky believed that human cognition operates at a higher level with symbols and structures (like frames and scripts), rather than just distributed patterns of activation. He often argued that connectionist models lacked a clear way to represent these symbolic structures.
  - Generalization and Abstraction: Minsky was concerned that connectionist models struggled with generalizing beyond specific training examples or abstracting high-level concepts from raw data.
  - Inefficiency: Minsky pointed out that many problems which seemed simple for symbolic models could be extremely computationally intensive for connectionist models.
  - Lack of explanation: Connectionist models, especially when they become complex, can be seen as "black boxes", making it difficult to interpret how they arrive at specific conclusions.

- Over-reliance on learning: Minsky believed that not all knowledge comes from learning from scratch, and some of it might be innate or structured in advance. He felt connectionism put too much emphasis on learning from raw data.

In 1980, there was a boom in expert knowledge systems that made AI recover interest. An **expert system** solves specific tasks following an ensemble of rules based on knowledge facilitated by experts. A remarkable use case was the XCON sorting system, developed for the Digital Equipment Corporation, which helped them save 40M$ per year. In addition, connectionism also came again on scene, thanks to the development of **backpropagation** applied to neurons, by Geoffrey Hinton. All these achievement made funding to come back to the field.

Nonetheless, there came a second winter of AI, from 1987 to 1994, mainly because several companies were disappointed and AI was seen as a technology that couldn't solve wide varieties of tasks. The funding was withdrawn from the field and a lot AI companies went bankrupt.

Luckily, from 1995 there started a new return of AI in the industry. The Moore's Law states that speed and memory of computer doubles every two years, and so computing power and memory was rapidly increasing, making the use of AI systems more feasible each year. During this time, many new concepts were introduced, such as **intelligent agents** as systems that perceive their environment and take actions which maximize their chances of success; or different **probabilistic reasoning tools** such as Bayesian networks, hidden Markov models, information theory, SVM,... In addition, AI researchers started to reframe their work in terms of mathematics, computer science, physics, etc., making the field more attractive for funding. A remarkable milestone during this time was the victory of Deep Blue against Garry Kasparov.

The last era of AI comes from 2011 to today, with the advent and popularization of **Deep Learning** (DL), which are deep graph processing layers mimicking human neurons interactions. This happened thanks to the advances of hardware technologies, that have enabled the enormous computing requirements needed for DL. The huge hype comes from the spectacular results shown by this kind of systems in a huge variety of tasks, such as computer vision, natural language processing, anomaly detection,...

In summary, we can see how the history of AI has been a succession of hype and dissapointment cycles, with many actors involved and the industry as a very important part of the process.

2 Machine Learning Basics

In this section, we review some notation, and basic knowledge of Linear Algebra, Probability and Machine Learning.

2.1 Linear Algebra Basics

A **scalar** is a number, either real and usually denoted $x \in \mathbb{R}$, or natural and denoted $n \in \mathbb{N}$. A **vector** is an array of numbers, usually real, $x \in \mathbb{R}^n$, or

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

A **matrix** is a 2-dimensional array of numbers, $A \in \mathbb{R}^{n \times m}$, or

$$A = \begin{bmatrix} A_{11} & \dots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{m1} & \dots & A_{mn} \end{bmatrix}.$$

A **tensor** is an n-dimensional array of numbers, for example $A \in \mathbb{R}^{m \times k \times p}$ is a 3-dimensional tensor.

Usually, we will be working with matrices, which can be operated in different ways:

- Transposition: A^T is the transposed of A, defined as $\left(A^T\right)_{ij} = A_{j,i}$.
- Multiplication: Let $A \in \mathbb{R}^{m\times k}, B \in \mathbb{R}^{k\times n}$, their multiplication, $C \in \mathbb{R}^{m\times n}$ is defined as

$$C = A \cdot B = AB = (C_{ij})_{i\leq m, j\leq n} = \left(\sum_k A_{ik}B_{kj}\right)_{i\leq m, j\leq n}.$$

  Note that the following holds for every matrix A, B:

$$(AB)^T = B^T A^T.$$

- Point-wise operations: if we have two matrices of the same size, $A, B \in \mathbb{R}^{m\times n}$, we can use apply scalar operator point-wise to each pair of elements in the same position in the two matrices. For example, the sum or the substraction of matrices.

There are also special matrices:

- Identity matrix: the identity matrix is a square matrix that preserves any vector it is multiplied with. For vectors of size n, the identity matrix I_n verifies

$$I_n x = x, \forall x \in \mathbb{R}^n.$$

- Inverse matrix: the inverse of a square matrix, $A \in \mathbb{R}^{n\times n}$, when it exists, is defined as the only matrix A^{-1} such that

$$A^{-1}A = AA^{-1} = I_n.$$

Another important concept is that of the norm, which is basically measuring how far a point is from the origin of the space and can be used to measure distances:

Definition 2.1. A **norm** is a function f that measures the size of vectors, and must have the following properties:

- $f(x) = 0 \iff x = 0$,
- $f(x+y) \leq f(x) + f(y)$, and
- $\forall \alpha \in \mathbb{R}, f(\alpha x) = |\alpha| f(x)$.

A very important family of norms is the L^p norm, defined as

$$\|x\|_p = \left(\sum_i |x_i|^p\right)^{\frac{1}{p}}.$$

The **Euclidean norm** is the L^2 norm, noted $\|x\|$ and equivalent to computing $\sqrt{x^T x}$. In Machine Learning, it is not uncommon to find the use of the squared Euclidean norm, since it maintains the ordinals and is easier to operate with. The **Manhattan norm** is the L^1 norm, and it is used when the difference between zero and nonzero elements is important. Finally, the **Max norm** is the L^∞, or $\|x\|_\infty = \max_i |x_i|$.

2.2 Probability Basics

A **random variable**, X, is a variable that can take different values, x, randomly. They can be **discrete**, like the number drawn from a dice, or **continuous**, like the humidity in the air.

A probability distribution, p, is a **Probability Mass Function (PMF)** for discrete variables, and a **Probability Density Function (PDF)** for continuous random variables. It must satisfy:

- The domain of p describe all possible states of X.
- $\forall x \in X, p(x) \geq 0$.
- $\int_{x \in X} p(x)\, dx = 1$.

It is usual to have two (or more) random variables, X and Y, and to be interested in the probability distribution of their combination, $p(x, y)$. In this context, we define the **marginal probability** of the variable X as

$$p(X = x) = \int_{y \in Y} p(x, y)\, dy, \forall x \in X.$$

The **conditional probability** of the variable Y conditioned to $X = x$ is

$$p(Y = y | X = x) = \frac{p(Y = y, X = x)}{P(X = x)}.$$

Finally, there is the **chain rule of conditional probabilities**, in which we start with n random variables, $X_1, ..., X_n$, and it follows:

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

Example 2.1. For example, let's say $X = \{1, 2, 3\}$, $Y = \{1, 2\}$ and $Z = \{1, 2\}$ with the following probabilities:

X	Y	Z	$p(x, y, z)$
1	1	1	$\frac{1}{6}$
1	1	2	$\frac{1}{6}$
1	2	1	$\frac{1}{12}$
1	2	2	$\frac{1}{24}$
2	1	1	$\frac{1}{24}$
2	1	2	$\frac{1}{20}$
2	2	1	$\frac{1}{20}$
2	2	2	$\frac{1}{20}$
3	1	1	$\frac{1}{6}$
3	1	2	$\frac{1}{12}$
3	2	1	$\frac{1}{20}$
3	2	2	$\frac{1}{20}$

Then, the marginal probabilities for the variable X are

$$P(X = 1) = \frac{1}{6} + \frac{1}{6} + \frac{1}{12} + \frac{1}{24} = \frac{11}{24},$$
$$P(X = 2) = \frac{1}{24} + \frac{1}{20} + \frac{1}{20} + \frac{1}{20} = \frac{23}{120},$$
$$P(X = 3) = \frac{1}{6} + \frac{1}{12} + \frac{1}{20} + \frac{1}{20} = \frac{21}{60} = \frac{7}{20}.$$

The conditional probability for the event $\{Y = 1 | X = 3\}$ is:

$$P(Y = 1 | X = 3) = \frac{P(Y = 1, X = 3)}{P(X = 3)} = \frac{\frac{1}{6} + \frac{1}{12}}{\frac{7}{20}} = \frac{\frac{1}{4}}{\frac{7}{20}} = \frac{5}{7}.$$

The conditional probability for the event $\{Z = 1 | X = 3, Y = 1\}$ is:

$$P(Z = 1 | X = 3, Y = 1) = \frac{P(X = 3, Y = 1, Z = 1)}{P(X = 3, Y = 1)} = \frac{\frac{1}{6}}{\frac{1}{4}} = \frac{2}{3}.$$

The probability of the event $\{X = 3, Y = 1, Z = 1\}$ could be computed from the conditional probabilities as follows, in case we only knew these:

$$\begin{aligned} P(X = 3, Y = 1, Z = 1) =& P(X = 3) \cdot P(Y = 1 | X = 3) \cdot P(Z = 1 | X = 3, Y = 1) \\ =& \frac{7}{20} \cdot \frac{5}{7} \cdot \frac{2}{3} = \frac{10}{60} = \frac{1}{6}. \end{aligned}$$

When there are several variables, it is possible that the value of one of them is dependant, somehow, on the values that the other variables take; or that it is not:

Definition 2.2. Two random variables X and Y are **independant**, denoted $X \perp Y$, if $\forall x \in X, y \in Y, p(X = x, Y = y) = p(X = x) \cdot p(Y = y)$.
X and Y are **conditionally independent** given the random variable Z, written $X \perp_Z Y$ if $\forall x \in X, y \in Y, z \in Z$,

$$p(X = x, Y = y | Z = z) = p(X = x | Z = z) \cdot p(Y = y | Z = z).$$

In Statistics and Machine Learning, there are some measures that summarize information about random variables, and that hold great importance.

Definition 2.3. The **expectation** of a function $f(x)$ where $x \sim p(x)$ is the average value of f over x:

$$\mathbb{E}_{x \sim p}[f(x)] = \int_{x \in X} p(x) f(x) \, dx.$$

The **variance** of $f(x)$ measures how the values of f varies from its average:

$$Var[f(x)] = \mathbb{E}\left[(f(x) - \mathbb{E}[f(x)])^2\right],$$

and the **standard deviation** is the square root of the variance.
The **covariance** of two random variables provides informaiton about how much two values are linearly related. More generally, if we apply two functions $f(x)$, where $x \sim p(x)$, and $g(y)$, where $y \sim p(y)$, the covariance between them is:

$$Cov[f(x), g(y)] = \mathbb{E}[(f(x) - \mathbb{E}[f(x)])(g(y) - \mathbb{E}[g(y)])].$$

2.3 Machine Learning Basics

To finalize with this review chapter, we are going to remember some basic concepts of Machine Learning.

First, let's give a definition of the concept:

Definition 2.4. A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

- The **task** T can be classification, regression, translation, generation, anomaly detection,...
- The **performance measure** P is specific to the tasks involved, and can be accuracy for classification, for example. It is measured on a **test set**.
- The **experience** E is divided into two main categories:
  - **Supervised learning**: a dataset of points associated with a label or a target determines the expected outcome of each event.

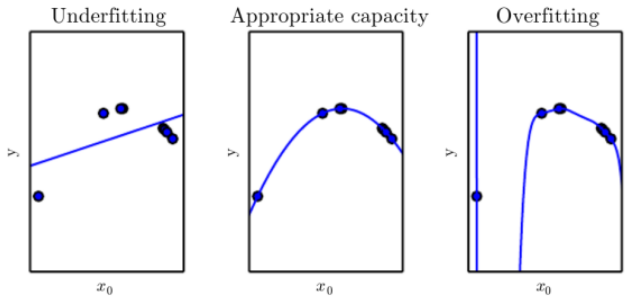


Figure 1: Appropriate capacity, overfitting and underfitting visualization.

– **Unsupervised learning**: a dataset of points without labels or targets, in which the desirable outcome needs to be define in some different way.

Mathematically, we can formalize this as having a dataset of m points and k features, which can be represented as a matrix $X \in \mathbb{R}^{m \times k}$. In the case of supervised learning, X is associated with a vector of labels, y, and we aim to learn a joint distribution, $p(X, y)$ to infer

$$p(Y = y|X = x) = \frac{p(x, y)}{\sum_{y'} p(x, y')}.$$

The goal is then to find a function $\hat{f}$ that associates each x to the best approximation of y, and that is capable of generalizing to unseen data. Usually, $\hat{f}$ is parameterized by a set of parameters, θ, which are learnt during training.

The main challenge of an ML model is **generalization** to unseen data estimated on test data after the training on training data. **Overfitting** occurs when the gap between training error and test error is too large, while **underfitting** occurs when the training error is too large. The **capacity** of a model is the range of functions that it is able to leanr and control how likely the model can overfit or underfit. This is visualized in Figure 1.

When we want to train a model, we will define the parameters that characterize it, and then we need to obtain the best possible of the parameters, according to the data. For this, we use estimators:

Definition 2.5. Given an unknown parameter θ, we estimate it through an **estimator**, $\hat{\theta}$. A **point estimator** is a function of the data, X,

$$\hat{\theta} = g(X).$$

The **bias** of an estimator is

$$bias\left(\hat{\theta}\right) = \mathbb{E}\left[\hat{\theta}\right] - \theta.$$

An estimator is **unbiased** if $bias\left(\hat{\theta}\right) = 0$.

The **variance** of an estimator is $Var\left(\hat{\theta}\right)$.

There are different ways to construct estimators, but one that is frequently used and that has solid mathematical foundations is the **maximum likelihood estimator**. Consider a dataset $X = \{x_1, ..., x_n\}$ and $p(x; \theta)$ a parametric family of probability distribution that maps for each x the probability $p_{data}(x)$. This is, for each θ, $p(x; \theta)$ is a probability density function. The maximum likelihood estimator is then

$$\begin{aligned}\theta_{ML} &= \arg\max_{\theta} p_{model}(X; \theta) \\ &= \arg\max_{\theta} \prod_{i=1}^{n} p_{model}(x_i; \theta),\end{aligned}$$

considering that all instances of data are independent and identically distributed (iid). It is also a common practice to use the maximum **log**-likelihood instead, removing the product and avoiding floating point issues, since when the dataset is large, the product will rapidly go to 0. In addition, the logarithm does not modify the ordinals of the function. Therefore, we can use:

$$\theta_{ML} = \arg\max_{\theta} \sum_{i=1}^{n} \log\left(p_{model}\left(x_i;\theta\right)\right).$$

3 Deep Neural Networks

3.1 Perceptron

A deeper explanation of the perceptron can be read in my notes from another course, https://lorenc1o.github.io/BDMA_Notes/universities/UPC/Machine_Learning_summary.pdf.

A perceptron is an algorithm for supervised learning of binary classifiers. That is, we have a dataset $X \in \mathbb{R}^{n\times m}$ associated with a vector of labels $y \in \{0,1\}^n$. Then, the perceptron learns a function $\hat{f}$ parametrized by a vector of weights $w \in \mathbb{R}^m$ and a bias b, such that:

$$\hat{f}\left(x\right) = \begin{cases} 1 & if\ w\cdot x + b > 0 \\ 0 & \sim \end{cases}.$$

Therefore, it is a linear classifier, which divides the input space into two regions separated by a hyperplane. This means that a perceptron cannot separate non-linear data.

3.2 Multi-layer perceptron

A deeper explanation of the MLP can be read in my notes from another course, https://lorenc1o.github.io/BDMA_Notes/universities/UPC/Machine_Learning_summary.pdf.

When we say 'Deep' neural network, we refer to a series of stacked perceptrons. However, just like this, the model is still linear. This is why activation functions are introduced. An **activation function** is a function that is applied to the output of a perceptron, to make it non linear.

For example, ReLU is a piecewise-linear function defined as

$$ReLU\left(z\right) = \begin{cases} z & if\ z \geq 0 \\ 0 & \sim \end{cases} = \max\left\{z, 0\right\}.$$

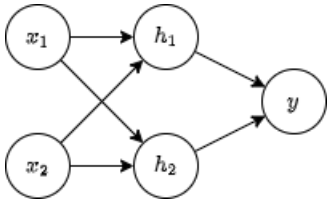
This function preserves much of the good oprimization properties of a linear function, i.e., it is differentiable (apart from one point), and its derivative is constant.

Example 3.1. Learn the XOR function with a 2-layer MLP.

The XOR function is represented with the table:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

We want to use a 2-layer MLP to learn this function:



In h_1, it will be

$$w_{11}x_1 + w_{12}x_2 + b_1,$$

and in h_2

$$w_{11}x_1 + w_{12}x_2 + b_1.$$

This can be represented as

$$h = W_h^T X + b_h.$$

Then, we apply ReLU

$$\max\left(0, W_h^T X + b_h\right),$$

and finally the output layer

$$y = W_y^T \max\left(0, W_h^T X + b_h\right) + b_y.$$

Let's see the different inputs:

x_1	x_2	h	y
0	0	$\begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$	$\begin{pmatrix} w_1^y & w_2^y \end{pmatrix} \begin{pmatrix} \max(0, b_1) \\ \max(0, b_2) \end{pmatrix} + b_y \leq 0$
0	1	$\begin{pmatrix} w_{12} + b_1 \\ w_{22} + b_2 \end{pmatrix}$	$\begin{pmatrix} w_1^y & w_2^y \end{pmatrix} \begin{pmatrix} \max(0, w_{12} + b_1) \\ \max(0, w_{22} + b_2) \end{pmatrix} + b_y > 0$
1	0	$\begin{pmatrix} w_{11} + b_1 \\ w_{21} + b_2 \end{pmatrix}$	$\begin{pmatrix} w_1^y & w_2^y \end{pmatrix} \begin{pmatrix} \max(0, w_{11} + b_1) \\ \max(0, w_{21} + b_2) \end{pmatrix} + b_y > 0$
1	1	$\begin{pmatrix} w_{11} + w_{12} + b_1 \\ w_{21} + w_{22} + b_2 \end{pmatrix}$	$\begin{pmatrix} w_1^y & w_2^y \end{pmatrix} \begin{pmatrix} \max(0, w_{11} + w_{12} + b_1) \\ \max(0, w_{21} + w_{22} + b_2) \end{pmatrix} + b_y \leq 0$

A solution is:

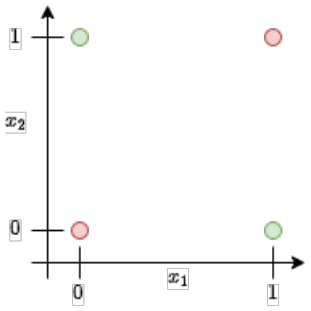
$$W_h = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, b_h = \begin{pmatrix} 0 \\ -1 \end{pmatrix}, W_y = \begin{pmatrix} 1 \\ -2 \end{pmatrix}, b = 0.$$

Let's check:

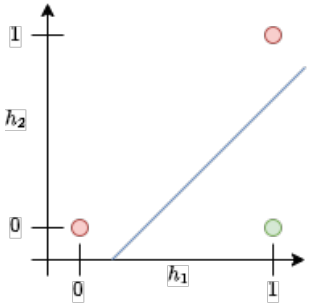
x_1	x_2	h	y
0	0	$\begin{pmatrix} 0 \\ -1 \end{pmatrix}$	$\begin{pmatrix} 1 & -2 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} = 0 \leq 0$
0	1	$\begin{pmatrix} 1 \\ 1-1 \end{pmatrix}$	$\begin{pmatrix} 1 & -2 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 1 > 0$
1	0	$\begin{pmatrix} 1 \\ 1-1 \end{pmatrix}$	$\begin{pmatrix} 1 & -2 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 1 > 0$
1	1	$\begin{pmatrix} 1+1 \\ 1+1-1 \end{pmatrix}$	$\begin{pmatrix} 1 & -2 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \end{pmatrix} = 0 \leq 0$

So, it works! Note that this solution is not unique!

What happens is actually that the solution for the XOR problem is not linearly separable:



But, the hidden layer transforms this space, making the problem linearly separable, and therefore solvable in the last layer:



3.3 Cost Functions

The cost function is important when working with neural networks, because our goal is to ultimately train the model to solve some problem, and the cost functions will be the function that our model will aim at minimizing, thus guiding the training process.

Usually, we will need to choose a cost function $\mathcal{L}$ that is suitable for our problem. Then, we will minimize $\mathcal{L}$ with stochastic gradient descent, by:

- Training on a training dataset.
- Estimating error on an evaluation dataset.
- Computing the gradients using backpropagation.

In this process, we will aim to find good local minima, instead of global minimum. This is related to overfitting (learning only the training data, losing generalization capabilities), and to the empirical fact that deep neural network have surprisingly good local and non-global optima.

3.3.1 Choice of cost function

In the general case, we use the maximum likelihood principle, taking the output types of the network into account. This means that we assume our dataset $\{x_1, ..., x_n\}$ to be independently and identically distributed (i.i.d.) from an unknown distribution, $p_{data}(x)$. We choose a parametric model family $p_{model}(x;\theta)$ represented as a neural network, which we use to estimate an approximation of the true distribution. For this, we utilize the **maximum likelihood estimator**, defined as

$$\theta_{ML} = \arg\max_{\theta} \prod_{i=1}^{n} p_{model}(x_i;\theta).$$

Usually, to avoid floating point errors, the log-likelihood is used instead:

$$\theta_{ML} = \arg\max_{\theta} \sum_{i=1}^{n} \log p_{model}\left(x_i; \theta\right).$$

In the maximum likelihood estimation framework, we might apply activation functions to the output layer to get a desired structure for our distribution. This choice will also influence the mathematical form of the cost function. For example, we can use linear units for regression or for Gaussian distributions, sigmoid units for binary classification or softmax units for multi-class classification.

Linear units for regression A **linear output layer** is such that, given the features h, the output is

$$\hat{y} = W^T h + b,$$

where W is the weights vector and b the bias.

We can use this to predict real or vector valued variables, such as prices, biometrics,...

Linear unit for Gaussian distribution A **Gaussian output unit** is such that, given features h, a linear layer produces a vector $\hat{y}$ representing the mean and the covariance matrix of a conditional Gaussian distribution:

$$p\left(y|x\right) = \mathcal{N}\left(y; \hat{y}, I\right).$$

Covariance is usually not modelled or simplified to be diagonal (in which case we need to ensure that the output is non-negative).

Binary classification In this case, the objective is to predict a binary variable, y: the neural network must predict $P\left(y = 1|x\right)$. Thus, we must ensure that the output is a probability, in the interval $[0, 1]$. For this, we can take

$$P\left(y = 1|x\right) = \max\left\{0, \min\left\{1, W^T h + b\right\}\right\}.$$

The problem with this approach is that if $W^T h + b \notin [0, 1]$ then the gradient is 0 and the training will stop. To solve this issue, we can use a **sigmoid unit**, which is

$$\hat{y} = \sigma\left(W^T h + b\right) = \frac{1}{1 + e^{-W^T h + b}}.$$

Softmax unit for multi-class classification Now our objective is to classify the input data into one among $N > 2$ classes. We want to predict $\hat{y}$ with $\hat{y}_i P\left(y = i|x\right)$, subject to $\hat{y}_i \in [0, 1], \forall i$ and $\sum_i y_i = 1$.

In the output layer we can have N perceptrons, each of them computing $z_i = \log P\left(y = i|x\right)$, i.e., the **logits**. With this, we can apply the **softmax output unit** to all of them, obtaining our vector of probabilities, as

$$softmax\left(z\right)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}.$$

3.3.2 Cross-entropy

In classification problems we want to estimate the probability of different outcomes. Let the estimated probability of outcome i be $p_{model}\left(x = i\right)$ with to-be-optimized parameters θ and let the frequency of outcome i in the training set be $p\left(x = i\right)$. Given N conditionally independent samples in the training set, then the likelihood of the parameters θ of the model $p_{model}\left(x = i\right)$ on the training set is:

$$\mathcal{L}\left(\theta\right) = \prod_{i \in X} p_{model}\left(x = i\right)^{N \cdot p(x=i)}.$$

Therefore, the log-likelihood, divided by N, is

$$\frac{1}{N}\log\left(\mathcal{L}\left(\theta\right)\right)=\frac{1}{N}N\cdot\sum_{i\in X}p\left(x=i\right)\log p_{model}\left(x=i\right)=\sum_{i\in X}p\left(x=i\right)\log p_{model}\left(x=i\right).$$

Cross-entropy minimization is frequently used in optimization and rare-event probability estimation. When comparing a distribution q against a fixed reference distribution p, cross-entropy and KL divergence are identical up to an additive constant (since p is fixed): According to the Gibbs' inequality, both take on their minimal values when $p=q$, which is 0 for KL divergence, and $H\left(p\right)$ for cross-entropy. In the engineering literature, the principle of minimizing KL divergence (Kullback's "Principle of Minimum Discrimination Information") is often called the Principle of Minimum Cross-Entropy (MCE), or Minxent.

3.4 Why deep NN?

Depth is the longest data path data can take from input to output. For a deep feed forward NN, depth is the number of hidden layers plus the output layer. State-of-the-art architectures used in practice have dozens to hundreds of layers.

Theorem 3.1. *Universal Approximation Theorem*
Let $\varphi\left(\right)$ be a nonconstant, bounded, and monotonically increasing continuous function. Let I_{m_0} denote the m_0-dimensional unit hypercube, $\left[0,1\right]^{m_0}$. The space of continuous functions on I_{m_0} is denoted by $C\left(I_{m_0}\right)$.
Then, given any function $f\in C\left(I_{m_0}\right)$ and $\varepsilon>0$, there exists an integer m_1 and sets of real constants α_i,b_i and $w_{ij}\in\mathbb{R}$ where $i=1,...,m_1$ and $j=1,...,m_0$ such that we may define

$$F\left(x\right)=\sum_{i=1}^{m_1}\alpha_i\cdot\varphi\left(\sum_{j=1}^{m_0}w_{ij}\cdot x_j+b_i\right)$$

as an approximate realization of the function f, that is

$$\left|F\left(x\right)-f\left(x\right)\right|<\varepsilon,\forall x\in I_m.$$

This theorem is very relevant, because it says that for any mapping function f in supervised learning, there exists a MLP with m_1 neurons in the hidden layer which is able to approximate it with a desired precision.

However, it only proves the existence of a shallow (just one hidden layer) MLP with m_1 neurons in the hidden layer that can approximate the function, but it does not tell how to find this number.

As a rule of thumb for the generalization error, it is

$$\varepsilon=\frac{VC_{dim}\left(MLP\right)}{N},$$

where VC_{dim} is the Vapnik-Chervonenkis dimension, a measure of the capacity of a model. It refers to the largest set of points that the model can shatter. It is not easy to compute, but a rough upper bound for a FFNN is $O\left(W\log W\right)$, with W being the total number of weight in the network.

Also, this theorem hints us that having more neurons in the hidden layers will give us better training error, but worse generalization error: overfitting.

However, for most functions m_1 is very high, and becomes quickly computationally intractable: so we need to go deeper.

Theorem 3.2. *No Free Lunch Theorem*
Multiple informal formulations:

- *For every learning algorithm A and B, there are as many problems where A has a better generalization error than problems where B has a better one.*
- *All learning algorithms ahve the same generalization error if we average over all learning problems.*
- *There is no universally better learning algorithm.*

Depth Property
The number of polygonal regions generated by a MLP with a ReLU function, d inputs, n neurons per hidden layer and l layers is

$$O\left(\binom{n}{d}^{d(l-1)} n^d\right).$$

This number grows exponentially with depth. This means that adding depth basically allows for more transformations of the input space.

3.5 Gradient-based Learning

The **gradient** of a function $f : \mathbb{R}^n \to \mathbb{R}$, is $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$, defined at the point $p = (x_1, ..., x_n)$ as

$$\nabla f(p) = \begin{pmatrix} \frac{\partial f}{\partial x_1}(p) \\ \vdots \\ \frac{\partial f}{\partial x_n}(p) \end{pmatrix}.$$

This is, it's the local derivative or slope of each dimension at a certain point.

Going in the opposite direction of the gradient is a naïve but practical guess of the direction of the local minimum. This is the base for the gradient descent method.

Gradient-descent method (Cauchy, 1847)
A parametric function $f(\theta)$ can be iteratively minimized by following the opposite direction of the gradient:

$$\theta_{t+1} = \theta_t - \varepsilon \nabla_\theta f(\theta),$$

where $\varepsilon > 0$ is the **learning rate**.
We stop iterating when the gradient is near to 0.

Notice that this is useless if we have a close form for the gradient! In that case it is easier to just minimize it. This is useful when this is not the case, which always happens for neural networks.

In addition, there are variations to the method, for example, we can vary ε during training.

Stochastic gradient descent
Given a cost function $f(\theta)$, parameters of the network are updated with

$$\theta \leftarrow \theta - \varepsilon \nabla_\theta f(\theta).$$

For the negative log-likelihood (MLE), the function is:

$$f(\theta) = \frac{1}{m} \sum_{i=1}^{m} L\left(x^{(i)}, y^{(i)}, \theta\right),$$

so the estimated gradient is

$$\nabla_\theta f(\theta) = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L\left(x^{(i)}, y^{(i)}, \theta\right).$$

The problem with this approach is that to take a single step of gradient descent, we must compute the loss over the whole dataset everytime, making the method not scalable at all. This is called **batch gradient descent**.

One solution is to compute the gradient with 1 sample only at each step, which is very noisy and inefficient, but works. This is the **stochastic gradient descent**.

In the middle ground, we find the **mini-batch gradient descent**, which divides the dataset into subsets, and updates the parameters after processing each of these subsets. A batch is a collection is a collection of samples used at each iteration for performing SDG in DL. A bigger batch provides a better gradient estimation, and therefore a faster learning, but also implies more device memory and slower descent.

Therefore, there is a tradeoff between money and performance at companies. In practice, a batch is set between 1 to 256 on one GPU.

But there is an even **greater problem**, the computation of the gradient is computationally very costly. To go around this problem, **back-propagation** was invented, as an efficient technique for gradient computation.

3.5.1 Back-propagation

The back-propagation algorithm is based on the chain rule for the derivative of composite functions: if we have $y = g(x)$ and $z = f(y) = f(g(x)) = (f \circ g)(x)$, then

$$\frac{df}{dx}(x) = f'(g(x))\, g'(x),$$

or, abusing notation,

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

This is generalized to multivariate funtions as follows: let $x \in \mathbb{R}^m, y \in \mathbb{R}^n, g : \mathbb{R}^m \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$. If $z = f(y) = f(g(x))$, then

$$\frac{\partial z}{\partial x_i}(x) = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i},$$

or,

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^T \nabla_y z,$$

where $\left(\frac{\partial y}{\partial x}\right)$ is the Jacobian of g.

Now, back-propagation is a recursive application of the chain rule, starting from the cost function. The algorithm works as follows:

1. **Forward pass**: a feedforward network takes as input x and produces the output $\hat{y}$. The information flows from layer to layer.

2. **Cost function**: compute the error between expected output and actual output.
3. **Back-propagate**: evaluate the individual gradient of each parameter and propagates them backwards to update them. For this, we use the concept of **local derivative**: the derivative of connected nodes are computed locally on the edges of the graph. For non-connected nodes, we multiply the edges connected between the nodes, and we sum over all incoming edges.

If we do this in a forward way, summing over all paths becomes intractable pretty quickly, while when doing it in a backwards way, it allows to obtain the derivative of the output with respect to every node directly in one pass. This leads to massive parallelization.

I did a more detailed explanation, with visualizations in my previous notes, https://lorenc1o.github.io/BDMA_Notes/universities/UPC/Machine_Learning_summary.pdf.

4 Deep Neural Networks: Optimization and Regularization

4.1 Optimization

Recall that neural networks learn by optimizing a cost function, $J(\theta)$. This optimization is, in practice, performed by gradient descent, so we must be able to compute $\nabla_\theta J(\theta)$. The problem is that this is computationally very costly. However, we have seen how back-propagation is an efficient gradient computation technique.

Now, learning is related to what we want to optimize, since we are interested in the performance on the test set, which is not always possible to ensure. Therefore, what is done is minimizing a cost function, J, hoping that it will improve the performance in the test set. But this relationship is also what makes learning and optimizing two different things! In pure optimization, our objective is minimizing J, while in learning, the objective is the ability to **generalize**, or perform well on the test set. Optimizing J on the training set does not ensure a good generalization, and sometimes worse results regarding the pure optimization problem in the training set, can yield better results in the learning problem (think about the overfitting problem).

Therefore, optimization is a crucial part of learning, and gradient-descent is a "cheap" optimization technique. However, it is not exempt of problems:

- Local minima and saddle points: small gradients can stop or greatly slow down the method.
- Partial estimation of gradients slows down descent, but can be beneficial for generalization. This refers to computing the gradient in batches, instead of in the full dataset (as we saw).

In addition, there are problems that are specific to the kind of functions that arise when working with neural networks:

- Bad convergence: due to the existence of many local optima.
- Long training time: the speed of stochastic gradient descent depends on initialization.
- Overfitting: deep neural networks have a lot of free parameters. They can sometimes learn by heart the whole training set, losing the ability to generalize.
- Vanishing gradients: in the backpropagation scheme, the first layers of the network may not receive sufficiently large gradients early in training.

There are different techniques to address these problems. Let's see some of them.

4.1.1 Solving Bad Convergence

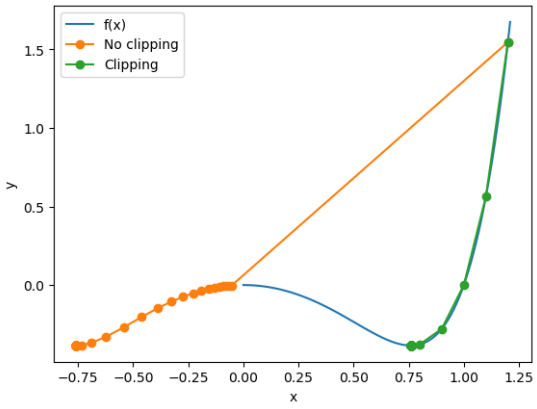
It's important to stabilize and improve the convergence of gradient descent on DNNs, and for this we need good optimization techniques.

Gradient Clipping Gradient Clipping proposes to clip the gradient norm to a threshold v, so:

1. Compute the gradient, say g.
2. If $\|g\| > th$, then
$$g \leftarrow \frac{g \cdot th}{\|g\|},$$
where th is a threshold to the maximum admissible gradient norm.

This way, we keep the direction of the gradient, while preventing *overshooting*, i.e., taking too large steps. Although this introduces a bias in the optimization process, it works well in practice.

A visualization is the following:



Here, we can observe how if we don't clip, the gradient is too large and we miss the minimum at the right, while clipping enables us to get ther easily.

Gradient with Momentum Momentum represents an acceleration method for stochastic gradient descent. The idea is to smooth gradient steps with some momentum or inertia, using previous gradient steps as a "*memory*" of the direction.

Let's call the gradient at step t, $G^t(\theta)$, then we define the velocity, $v^t(\theta)$, as

$$v^t(\theta) = \alpha v^{t-1}(\theta) - (1-\alpha) G^t(\theta),$$

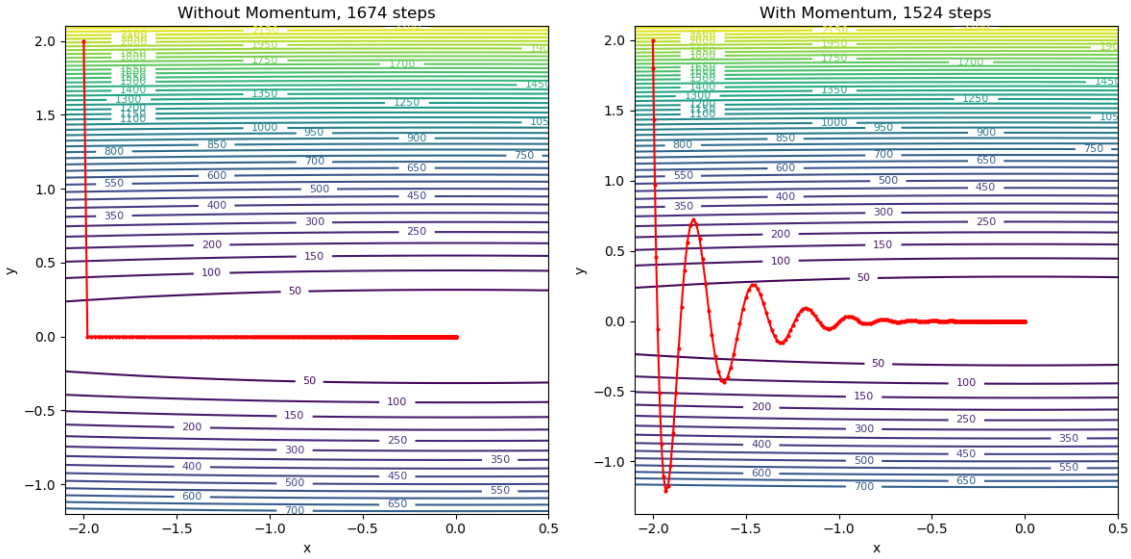
where $0 \leq \alpha < 1$ is a parameter controlling how much of the gradient we use for the parameter update, usually set as 0.9. If it is set to 0, we obtain the vanilla SGD.

The updates are done as

$$\theta^{t+1} = \theta^t - \eta v^t(\theta),$$

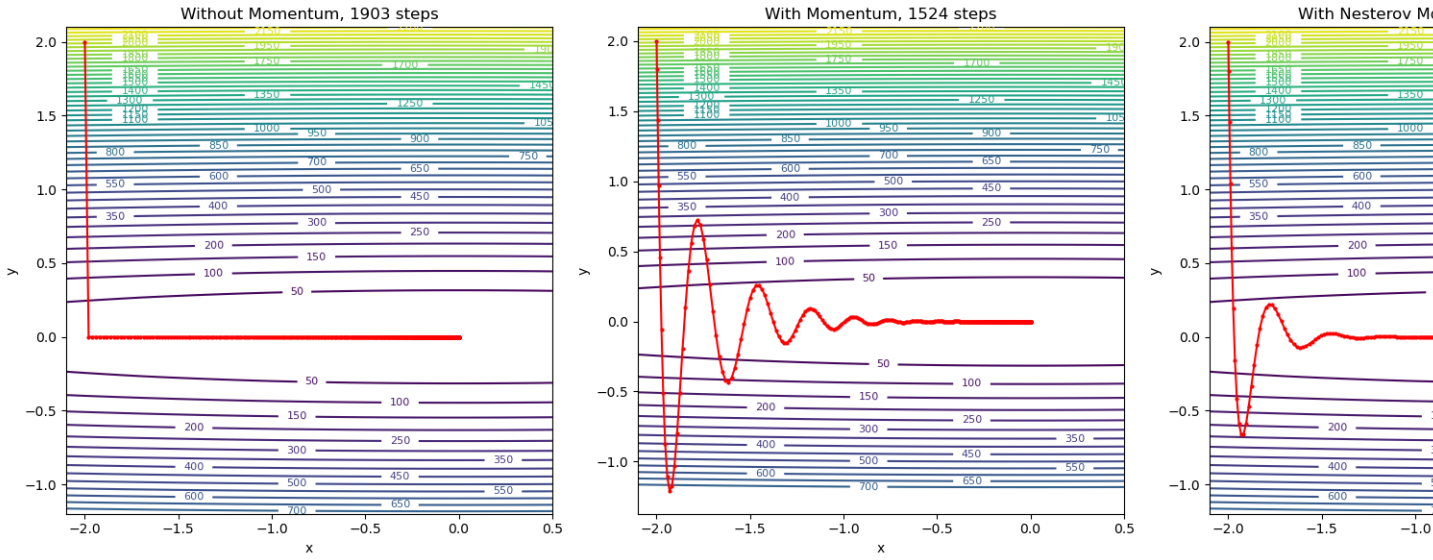
where η is the learning rate.

The momentum approach is shown below, observing a slight speedup.



Nesterov Momentum This represents a variant of gradient descent with momentum. It tries to address the problem of the momentum approach, which is that is tends to oscillate around the minimum. Nesterov corrects these oscillations by estimating the gradient after the momentum update as:

$$v^t(\theta) = \alpha v^{t-1}(\theta) - (1-\alpha)\, G\left(\theta - \alpha v^{t-1}(\theta)\right).$$



We can observe how the oscillations are less prominent. If the function was different and momentum oscillates around the minimum, Nesterov would help with this.

Dynamic Learning Rate Stochastic gradient descent estimates the gradient with only one sample, and iterates over the whole dataset, as

$$\theta \leftarrow \theta - \eta G\left(\theta; x^i, y^i\right).$$

It is common to use minibatches instead:

$$\theta \leftarrow \theta - \eta G\left(\theta; x^{i_0:i_0+B}, y^{i_0:i_0+B}\right) = \theta - \eta \nabla_\theta \sum_{i=i_0}^{i_0+B} L\left(\theta; x^i, y^i\right).$$

The learning rate is very important, and in fact it should decrease during training, as we get closer to the optimum. Some reasons to make this decrease are:

- True gradients becomes small when θ is close to the minimum.
- With SGD, estimating the gradient with samples introduces noise, and these gradients don't necessarily decrease.
- A sufficient condition for convergence of SGD is:

$$\sum_{k=1}^{\infty} \eta_k = \infty, \ and \ \sum_{k=1}^{\infty} \eta_k^2 < \infty.$$

In practice, what we do is apply a linear decay until some point τ, and keep it constant after this point:

$$\eta_k = \left(1 - \frac{k}{\tau}\right) \eta_0 + \frac{k}{\tau} \eta_\tau,$$

and η_τ after τ iterations.

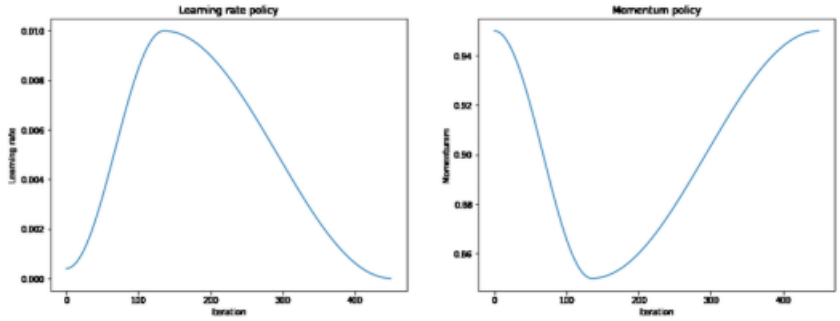
Now, the question is, how to choose η_0, η_τ and τ? There are several options:

- With trail and error (using train/validation sets).
- It's better to monitor the loss function during training.
- A practical choice:
  - Choose τ so that the whole training dataset is seen around 100 times.
  - Choose η_τ around 1% of η_0.
  - η_0 comes with experience:
    - Too big: big variations in loss
    - Too small: learning is slow, we can get stuck in a plateau
  - Recipe:
    - Try multiple η_0 over 100 iterations
    - Pick η_0 slightly higher than the best

Cyclical Learning Rate Optimization difficulties comes more from plateaus than from bad local optima, and increasing η allows to go across these plateaus. For this, what is done is varying the learning rate in a cyclical manner, from a minimum bound, η_{min} to a maximum bound, η_{max}.

Super Convergence with One-Cycle Policy Only once cycle works as:

- Start with a small learning rate to begin convergence.
- Increases and then stabilizes to high value, to cross big plateaus.
- Decreases to a small value, to optimize local minima.



SGD with Adaptive Learning Rates Preconditioning:

$$\theta^{t+1} \leftarrow \theta^t - \eta_t P_t^{-1} G\left(\theta^t\right),$$

where P_t can be defined in different ways. Let's see AdaGrad and RMSProp.

AdaGrad: parameters with largest partial derivatives should have a rapid decrease.

$$P_t = \left[diag\left(\sum_{j=0}^{t} G\left(\theta^t\right) G\left(\theta^t\right)^T \right) \right]^{\frac{1}{2}}.$$

More precisely:

$$\begin{cases} g^t \leftarrow G\left(\theta^t\right) \\ r^t \leftarrow r^{t-1} + g^t \odot g^t \\ \theta^{t+1} \leftarrow \theta^t - \frac{\lambda}{\delta + \sqrt{r^t}} \odot g^t \end{cases}$$

AdaGrad converges quickly on convex problems. However, keeping all the history with momentum can be detrimentary, and smoothing the gradient destroys information.

RMSProp: introduces momentum when computing the precondicioner. The idea is to adapt the learning rate to the curvature of the loss function, putting the brakes on when the function is steep and accelerating when the loss function is flat.

$$P_t = \left[diag\left(\alpha P_{t-1} + \sum_{j=0}^{t} G\left(\theta^t\right) G\left(\theta^t\right)^T \right) \right]^{\frac{1}{2}}.$$

More precisely:

$$\begin{aligned} v^t(\theta) &= \alpha v^{t-1}(\theta) + (1-\alpha) G(\theta)^2 \\ \theta^{t+1} &= \theta^t - \frac{\eta}{\epsilon + \sqrt{v^t(\theta)}} G(\theta). \end{aligned}$$

Adam Adam (Adaptative Moment Estimation) builds on RMSProp, but also uses a moving average of the gradients. It works as:

$$\begin{aligned} m^t(\theta) &= \beta_1 m^{t-1}(\theta) + (1-\beta_1) G(\theta) \\ v^t(\theta) &= \beta_2 v^{t-1}(\theta) + (1-\beta_2) G(\theta)^2 \\ \theta^{t+1} &= \theta^t - \eta \frac{m(\theta)}{\epsilon + \sqrt{v(\theta)}}. \end{aligned}$$

In practice, Adam is the most used optimizer. However, there are more efficient algorithm, like LARS (Layerwise Adaptive Rate Scaling) or LAMB (LARS+Adam).

Comparison

- SGD momentum should allow for better solution, but hyperparameters are harder to find.
- Adam is easier to tune.

4.2 Initialization and Normalization

The parameters of a deep learning model need initial values. The assignation of initial values is called **initialization**, and can impact the optimization process in several ways:

- A bad initialization can make the training process not to converge.
- It can impact the convergence quality, in terms of speed and the value reached.
- Also, it can impact the generalization error.

Therefore, a difficult question arises: initial parameters can help optimization, but can detriment the generalization error.

> **Principle: Break Symmetries**
> Two identical parameters, connected to the same input, should be initialized differently, to incentivize different learning.

One option could be to initialize everything to 0, but this is not a good choice, because it disentivizes learning. Instead, there are different initialization schemes, like random initialization.

4.2.1 Random Initialization

Random initialization uses a probability distribution to initialize the weights. For example, **Xavier initialization** uses an uniform distribution as

$$W_{i,j} \sim U\left(-\frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}, \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}\right),$$

where n_j is the number of neurons in layer $j = 1, ..., N$. The input layer is initialized as

$$W_{0,j} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_1}}\right).$$

4.2.2 Input normalization

Gradient descent is sensitive to strong variations in the input, and, ideally, the surface of the loss function should ahve a uniform curvature in all directions, similar to a sphere. This can be incentivized by input normalization, i.e., normalizing all input parameters so that they all lie in a similar value range.

Moreover, there is the concept of **batch normalization**, or adaptive reparametrization. This consists in normalizing the input of each layer of a neural network. It is motivated by the following:

- Deep neural networks are compositions of functions, whose parameters are iteratively updated during training.
- The updates are done simultaneously to all layers, and unexpected effects can come into play, since each layer is updated assuming all other layers remain constant.

- Therefore, the updates to other layers can add high order effects that can lead to the problem of **gradient explosion**, a situation in which the gradient keeps growing indefinitely.

In this case, the idea is to normalize the distribution of each input feature in each layer, across each minibatch, to a normal, $\mathcal{N}(0,1)$:

$$\mu \leftarrow \frac{1}{m}\sum_{i=1}^{m}\overline{x}^i,$$
$$\sigma^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}\left(\overline{x}^i-\mu\right)^2,$$
$$\overline{x}^i \leftarrow \frac{\overline{x}^i-\mu}{\sqrt{\sigma^2-\varepsilon}}.$$

Remember that ε is the approximation of the generalization error, $\varepsilon = \frac{VC_{dim}}{N}$.

4.3 Regularization

The loss function can be small on the training data, but large on the dataset. This is the overfitting problem, that we have seen before. Deep NN can tend to overfit, since more depth implies more free parameters, and therefore a higher VC dimension, which can increase ε according the rule of thumb for its approximation.

A way to go around this is to put constraints on the weights, to reduce the VC dimension. This can be done through regularization.

4.3.1 $\mathcal{L}_2$ Regularization (or Ridge)

$\mathcal{L}_2$ regularization keeps the $\mathcal{L}_2$ norm of the free patameters, $\|\theta\|$, as small as possible, during learning.

The intuition is that each neuron will use all its inputs with small weights, instead of specializing on a small part with high weights.

To accomplish this, we have to minimize two things at the same time: the training loss and a penanlty term representing the norm of the weights:

$$\mathcal{J}(\theta) = \mathbb{E}_D\left[\|t-y\|^2\right] + \lambda\|\theta\|^2,$$

where λ is the **regularization parameter**, controlling the strength of the regularization:

- If λ is small, there is only a small regularization, allowing higher weights.
- If λ is high, the weights will be kept very small, but they may not minimize the training loss.

The gradient of this new loss function is

$$\nabla_\theta\mathcal{J}(\theta) = -2(t-y)\nabla_\theta y + 2\lambda\theta,$$

and so the parameter updates become

$$\Delta\theta = \eta(t-y)\nabla_\theta y - \eta\lambda\theta.$$

The $\mathcal{L}_2$ regularization leads to weights decay: even if there is no output error, the weight will converge to 0, forcing the weights to constatly learn, and disincentivizing the specialization on particular examples (overfitting), enhancing generalization.

4.3.2 $\mathcal{L}_1$ regularization (or Lasso)

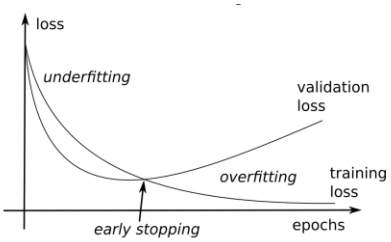
In this case, we penalize the absolute value of the weights, instead of their euclidean norm:

$$\mathcal{J}(\theta) = \mathbb{E}_D\left[\|t - y\|^2\right] + \lambda \|\theta\|_1 .$$

This method leads to very sparse representations, where a lot of neurons may be inactive, and only a few represent the input.

4.3.3 Early Stopping

During training, a common behavior is the following:



It's usual that the training error decreases constantly, while the validation error decreases, and the increases again. If we manage to find the optimal point in this process, we can stop earlier the training process, before the validation error gets larger.

This method is equivalent to $\mathcal{L}_2$ normalization, both limiting the capacity of the model:

- With Ridge regularization, small slope regions contract the dimension of θ, which decays to 0, and high slope regions are not regularized because they help descent.
- With early stopping, parameters with high slope are learned before parameters with low slope.

4.3.4 Dropout

Dropout considers all the networks can be formed by removing some units from a network. With this in mind, the method consists of:

- At each optimization iteration: we apply random binary masks on the units to consider.

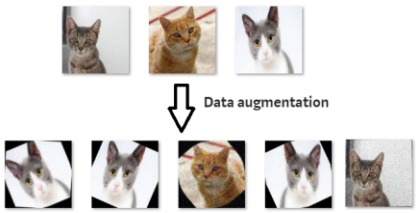
The probability of dropout, p, is a hyperparameter. Therefore, what we do at training time is, at each step, deactivate some neurons, randomly. This incentivizes generalization, since relying on some neurons specializing a lot for some features of the training data is harder if these neurons are not always available.

Note that, at inference time, all neurons are always available.

4.3.5 Data Augmentation

The best way to avoid overfitting is collecting more data, but this can be hard, costly, or simply impossible. A simple trick is **data augmentation**, which consists in creating new varied data from the current data by perturbing it, while not chaning the labels associated to it.

For example:



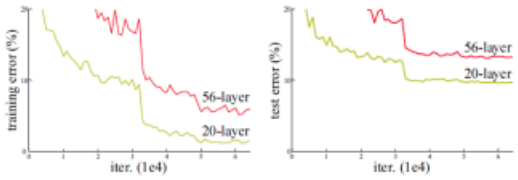
Mixup is a particular case of data augmentation, consisting on creating new training new samples with labels by interpolation:

$$\hat{x} = \lambda x_i + (1 - \lambda)\, x_j,$$
$$\hat{y} = \lambda y_i + (1 - \lambda)\, y_j,$$

where $\lambda \sim Beta\,(\alpha, \alpha)$, with $\alpha \in [0.1, 0.4]$ for classification tasks. This method works for structured data, and can be used to stabilizing GANs (we will see this later).

4.4 Vanishing Gradients

Contrary to what we could think, adding more layers to a DNN does not necessarily lead to better performance, both on the training and test set. For instance, see the following graph, where we observe the performance of a 20 layers NN (left) and a 56 layers NN (right):



We observe how its performance is worse in all senses. The main reason for this is the **vanishing gradient problem**. The gradient of the loss function is repeatedly multiplied by a weight matrix W as it travels backwards in a deep NN, as

$$\frac{\partial h_k}{\partial h_{k-1}} = f'\left(W^k h_{k-1} + b^k\right) W^k.$$

When the gradient arrives to the first layer, the contribution of the weight matrices is comprised between W^d_{min}, and W^d_{max}, which are the weight matrix with the highest and lowest norm, and d is the depth of the network. We find:

- If $|W_{max}| < 1$, then W^d_{max} is very small for high values of d, and the gradient vanishes.
- If $|W_{min}| > 1$, then W^d_{min} is very high for high values of d, and the gradient explodes.

We saw that exploding gradients can be solved by gradient clipping. But vanishing gradients are still the current limitation of deep NN. The solutions include the utilization of ReLU activaiton functions, unsupervised pre-training, batch normalization, residual networks, etc.

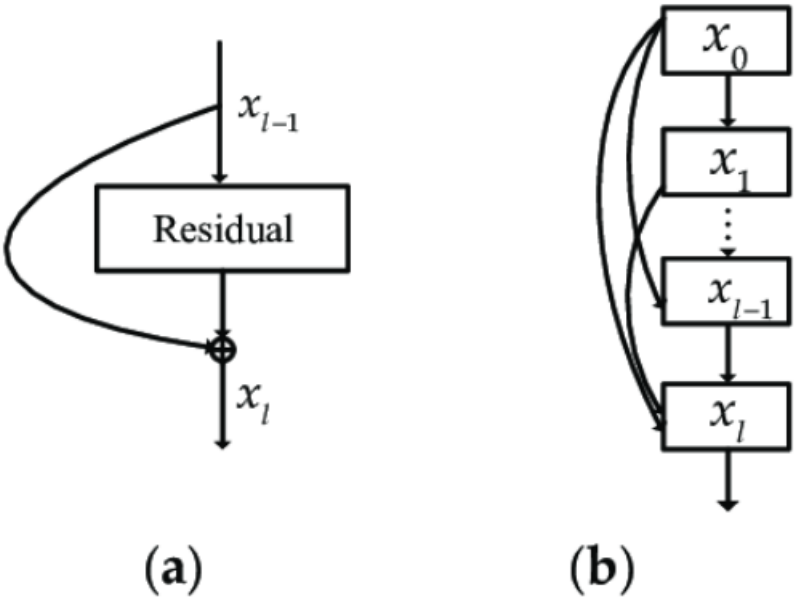
4.4.1 Residual network

A Residual Neural Network, or **ResNet**, is an advanced type of neural network that is specifically designed to help improve the performance of deep learning models.

ResNet introduces the concept of residual learning. Instead of expecting each stack of layers to directly fit a desired underlying mapping, ResNet layers are designed to fit a residual mapping. The key component of ResNet

is the introduction of "skip connections" or "shortcuts" that bypass one or more layers. A skip connection in a ResNet allows the gradient to be directly backpropagated to earlier layers.

These skip connections perform identity mapping, and their outputs are added to the outputs of the stacked layers. This design helps in training deeper networks by mitigating the vanishing gradient problem. With residual blocks, the network learns the additive residual function with respect to the layer inputs, making it easier to optimize and gain accuracy from considerably deeper networks.



4.4.2 Stochastic Depth

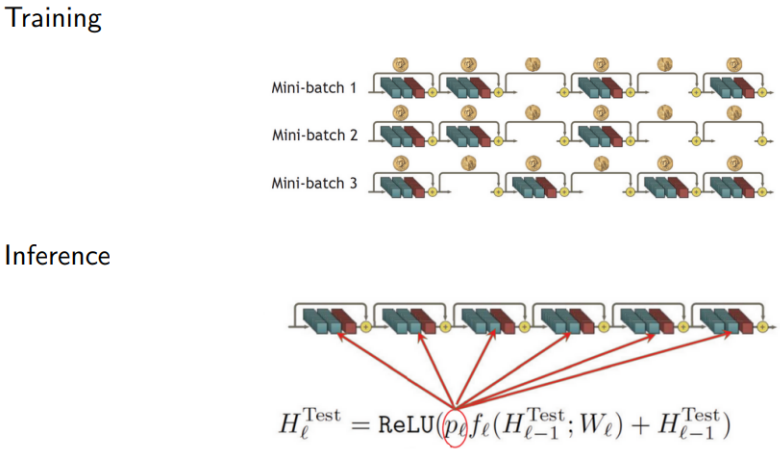
Stochastic depth is a training technique for deep neural networks, particularly effective for very deep networks like ResNets. It was introduced as a solution to the problem of vanishing gradients and long training times in deep networks.

During training, stochastic depth randomly drops layers in the network. The idea is similar to dropout, where random neurons are turned off during training to prevent overfitting. In stochastic depth, however, it's entire layers that are dropped.

Each training iteration uses a shallower version of the network. The depth of the network varies each time, as different subsets of layers are randomly deactivated.

Notice how stochastic depth is particularly useful for ResNets, where skip connections (or residual connections) are a key feature. When a residual block is dropped, the skip connection effectively takes its place, allowing the signal to still propagate forward.

At test time, all layers are used, but their outputs are scaled appropriately to compensate for the dropout during training. This ensures that the network can benefit from its full depth during inference.



4.5 Double Descent

Double descent is a phenomenon observed in the training of machine learning models, particularly in relation to the model's complexity and its performance on a given task. This concept challenges the traditional understanding of the bias-variance tradeoff and has gained attention in the field of machine learning and statistics.

The double descent curve shows that after the point where the model starts to overfit (as per the traditional U-shaped bias-variance tradeoff curve), increasing the model complexity even further can lead to a decrease in the total error again.

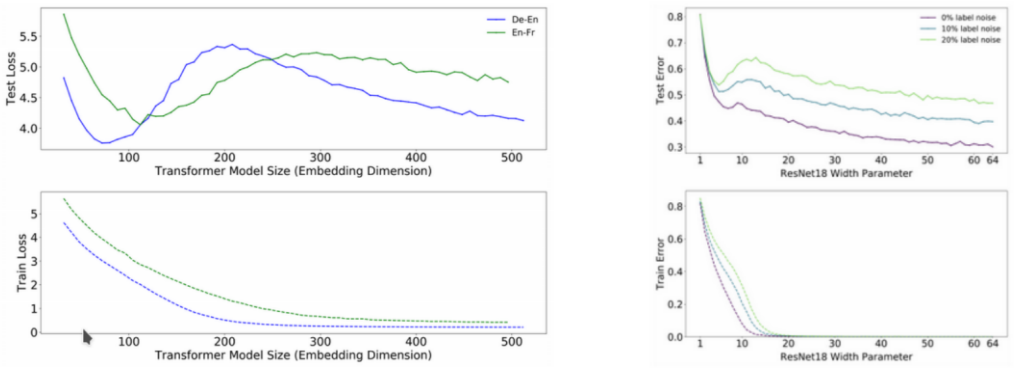
It shows the following phases:

- **Underparameterized Regime**: Where the model has too few parameters and underfits the data. Here, increasing complexity reduces bias and total error.
- **Interpolation Threshold**: At this point, the model just starts to fit all the training data perfectly (including noise), leading to high variance and total error.
- **Overparameterized Regime**: Beyond this threshold, as complexity continues to increase, the model enters the second descent where surprisingly, the total error begins to decrease again despite the model being overparameterized.

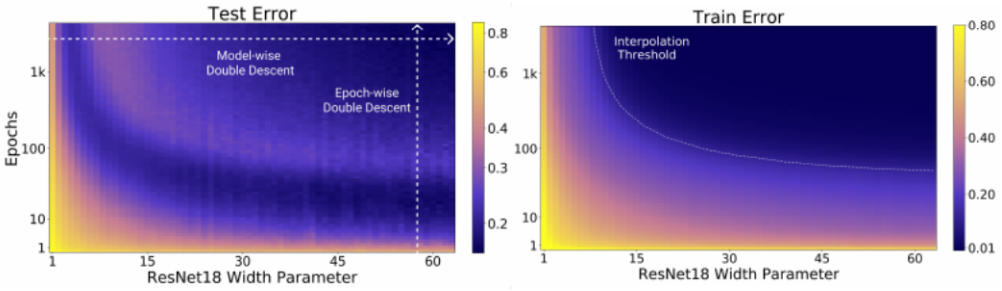
The double descent phenomenon is especially noticeable in scenarios with limited training data. With more data, the peak of the curve (at the interpolation threshold) becomes less pronounced.

Double descent suggests that in some cases, choosing an even more complex model after hitting the overfitting point can improve performance.

For example, observe this phenomenon in the following graphs:



A similar phenomenon is **grokking**, which occur when we increase training time. It is remarkable that shallow models don't show it, which is a reason to use deep networks!



Part II

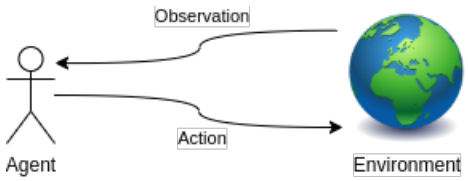
Reinforcement Learning

5 Introduction

People and animals learn by interacting with the environment that sorround them, differing from certain other types of learning. This process is active, rather than passive: the subject needs to perform interactions with the environment, to obtain knowledge. Also, the interactions are usually sequential, with future interactions possibly depending on earlier ones.

Not only this, but we are goal oriented: we act towards an objective. And, more importantly, we can learn without examples of optimal behavior! Instead, we optimise some reward signal obtained from the outcome of our actions.

It is in these observation that **Reinforcement Learning (RL)** arises as a learning paradigm, based on the **interaction loop**: there is an agent in an environment; the agent can make actions in the environment, and get observations from it.



RL relies on the reward hypothesis:

Conjecture 5.1. *Reward Hypothesis*
Any goal can be formalized as the outcome of maximizing a cumulative reward.

This hypothesis basically says that every objective that an agent can have, can be stated in terms of maximizing a reward associated to the actions of the agent with respect to this objective.

For example, if the objective of the agent is to fly a helicopter from point A to point B, then the reward could be negatively affected by the distance to point B, by the time taken to reach B,...

Now, it is important to realize that there exist different reasons to learn:

- Find solutions to problems.
- Adapt online to unforseen circumstances.

Well, RL can provide algorithm for both cases! Note that the second point is not just about generalization, but also to cope with the so-called data shift, efficiently, during operation.

With all this, now we can define RL:

Definition 5.1. Reinforcement Learning is the science and framework of learning to make decisions from interaction.

This requires us to think about time, consequences of actions, experience gathering, future prediction, uncertainty,...

It has a huge potential scope and is a formalisation of the AI problem.

6 Definition and components

Definition 6.1. The **environment** is the world of the problem at hand, with **agents** in it that can perform actions, over time.
At each time step t, the agent:

- Receives observation O_t and reward R_t from the environment.
- Executes action A_t.

And the environment:

- Receives action A_t.
- Emits observation O_{t+1} and reward R_{t+1}.

But, what is a reward?

A **reward**, R_t, is a scalar feedback signal which indicates how well the agent is doing at step t: it defines how well the goal is being accomplished!

Therefore, the agent's job is to maximize the cummulative reward of the future steps, i.e.,

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ...$$

This is called the **return**. But, when one thinks about it carefully, one realizes that it is hard to know the future rewards with such precision. Therefore, it is also usual to use the **value**, which is the expected return, taking into account the current state, s:

$$v(s) = \mathbb{E}[G_t | S_t = s].$$

This depends on the actions the agents takes, and the goal is to maximize it! To achieve this, the agent must pick suitable actions.

Therefore, rewards and values define the utility of states and actions, and in this setup there is no supervised feedback.

Note, also, that this values can be defined recursively as

$$G_t = R_{t+1} + G_{t+1},$$

$$v(s) = \mathbb{E}[R_{t+1} + v(S_{t+1}) | S_t = s].$$

The **environment state** is the environment's internal state, which is usually invisible or partially visible to the agent. It is very important, but it can also contain lots of irrelevant information.

An environment is **fully observable** when the agent can see the full environment state, so every observation reveals the whole environment state. That is, the agent state could just be the observation:

$$S_t = O_t.$$

Note that S_t is the agent state, not the environment state!

6.1 Maximising the value by taking actions

As we have outlined, the goal is to select the actions that maximise the value. For this, we may need to take into account that actions may have long term consequences, delaying rewards. Thus, it may be better to sacrifice immediate reward to gain long-term reward.

The decision making process that for a given state chooses which action to take is called a **policy**.

To decide which action to take, we can also condition the value on actions:

$$q(s,a) = \mathbb{E}[G_t | S_t = s, A_t = a],$$

so, for a given state s, a possible set of actions A_t^s, we could decide which action to take as

$$a_t = \max_{a \in A_t^s} q(s,a).$$

Then, the **history** is the full sequence of observation, actions and rewards:

$$\mathcal{H}_t = O_0, A_0, R_1, O_1, ..., O_{t-1}, A_{t-1}, R_t, O_t.$$

6.2 Markov Decision Processes

Markov Decision Processes (MDPs) are a useful mathematical framework, defined as:

Definition 6.2. A decision process is Markov if

$$p(r, s | S_t, A_t) = p(r, s | \mathcal{H}_t, A_t).$$

This means that the current state is the only information needed to make a decision, we don't need the full story. For example, think in a chess game: there are many ways to arrive to a certain position, but it really does not matter how to got to the position, the past does not affect your choice now.

In order for a process to be Markov, full observability is required. When the situation is of **partial observability**, the observations are not Markovian, so using the observation as state is not enough to make the decision. This is called a **partially observable Markov decision process (POMDP)**. Note that the environment state can still be Markov, but the agent does not know it. In this case, we might be able to construct a Markov agent state.

In the general case, the agent state is a function of the history:

$$S_{t+1} = u(S_t, A_t, R_{t+1}, O_{t+1}),$$

where u is a **state update function**.

Usually, the agent state is much smaller than the environment state.

Example 6.1. A not Markov process:

Consider the following maze to be the full environment:

And consider the following observations:

They are indistinguishable! This process is not Markov, because only taking into account the current state, we cannot identify where we are.

To deal with partial observability, agent can construct suitable state representations. Some examples of agent states are:

- Last observation: $S_t = O_t$ (might not be enough).
- Complete history: $S_t = \mathcal{H}_t$ (might be too large).
- A generic update: $S_t = u(S_{t-1}, A_{t-1}, R_t, O_t)$ (but how to design u?)

Constructing a fully Markovian agent state is often not feasible and, more importantly, the state should allow for good policies and value predictions.

6.3 Policies

As we saw, a **policy** defines the agent's behavior: it is a map from the agent state to an action. Policies can be deterministic,

$$A = \pi(S),$$

or stochastic,

$$\pi(A|S) = p(A|S).$$

6.4 Value Functions

We saw the value before, which is the expected return. However, it is usual to introduce a **discount factor**, $\gamma \in [0, 1]$, which trades off importance of immediate and long-term rewards. This way, the value becomes

$$v_\pi(s) = \mathbb{E}[G_t|S_t = s, \pi] = \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|S_t = s, \pi\right].$$

The value depends on the policy, π, and can be used to evaluate the desirability of states, as well as to select between actions.

Note the role of the discount factor: the higher it is, the higher the focus on long term outcomes.

Now, using the recursive expression of the return, $G_t = R_{t+1} + \gamma G_{t+1}$, we can rewrite the value as

$$v_\pi(s) = \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = a, A_t \sim \pi(s)\right],$$

where $A \sim \pi(s)$ means A is chosen by policy π in state s. This is known as a **Bellman equation**. A similar equation holds for the optimal value, i.e., the highest possible value:

$$v_*(s) = \max_a \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a\right].$$

Note how this does not depend on a policy, it is just the maximum achievable value from the current state.

6.4.1 Value Function Approximations

Agents often approximate value functions, and with an accurate value function approximation, the agent can behave optimally, or very well, even in intractably big domains.

6.5 Model

A **model** predicts what the environment will do next. For example, $\mathcal{P}$ predicts the next state, given the current state and an action:

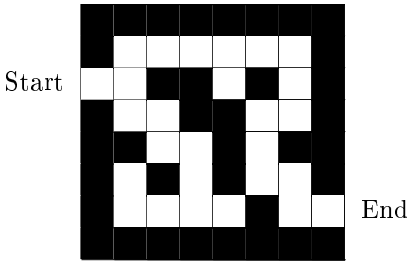
$$\mathcal{P}(s, a, s') \approx p(S_{t+1} = s' | S_t = s, A_t = a).$$

Or $\mathcal{R}$ predicts the next immediate reward:

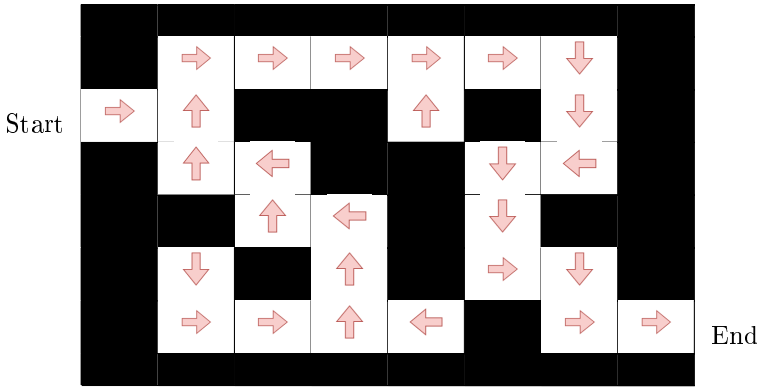
$$\mathcal{R}(s, a) \approx \mathbb{E}[R_{t+1} | S_t = s, A_t = a].$$

Note that a model does not immediately give us a good policy! We still need to plan and see how actions and states are related.

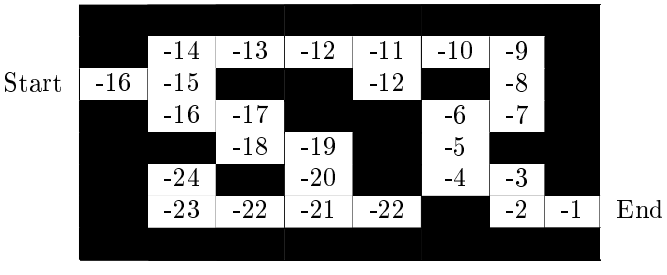
Example 6.2. Consider the following maze, where the rewards are -1 per time-step, the actions are to go N, E, S and W, and the states are the agent's location:



The following arrows represent the policy, $\pi(s)$, for each state s:



In the following one, the numbers represent the value $v_\pi(s)$ of each state s:



The grid layout represents the partial transition model $\mathcal{P}^a_{ss'}$, and numbers represent the immediate reward, $\mathcal{R}^a_{ss'}$ from each state s, which is -1 for all a and s' in this case.

6.6 Agent categories

An agent is **model free** when the behavior of the environment is not known. The agent needs a policy or a value function to operate and there is no model. On the other hand, it is **model based** when the environment is known by means of a model. In this case, a policy and a value function might be optional, since it is possible that the agent can operate just knowing the model.

Model free agents are simpler, while model based agents are more sample efficient.

Another categorization is the following:

- Value based: there is no policy, it is implicit in the value function.
- Policy based: there is no value function, the model operates only by means of the policy.
- Actor critic: they have both a policy and a value function.

6.7 Subproblems of RL

Prediction consists in evaluating the future, for a given policy, i.e., what are the values in each state?

Control refers to the problem of optimising the future to find the best policy, i.e., which actions to take?

These two problems are strongly related, because the best actions to take will be decided using our predictions about the future:

$$\pi_* (s) = \arg\max_{\pi} v_\pi (s) .$$

Two fundamental problems in RL are:

- Learning: the environment is initially unknown and the agent interacts with it to learn.
- Planning/search: a model of the environment is given or learnt, and the agent plans in this model.

In order to learn, we need to define all components of the problem as functions:

- Policy: $\pi : S \to A$ (or probabilities over A).
- Value functions: $v : S \to \mathbb{R}$.
- Models: $p : S \to S$ or $r : S \to \mathbb{R}$.
- State update: $u : S \times O \to S$.

Then, we can use, for example, neural networks and deep learning techniques to learn. But we do need to be careful, because in RL it is usual to violate assumptios made in supervised learning, such as having i.i.d. samples, or stationarity.

7 Markov Decision Processes

We saw the notion of MDP, and now we formalize it:

Definition 7.1. A **Markov Decision Process (MDP)** is a tuple (S, A, p, γ) where:

- S is the set of all possible states with the Markov Property.
- A is the set of all possible actions.
- $p(r, s'|s, a)$ is the joint probability of a reward, r, and next state, s', given a state s and an action a.
- $\gamma \in [0, 1]$ is a discount factor that trades off later rewards to earlier ones.

Remark 7.1. p defines the dynamics of the problem.

Sometimes, it is useful to marginalise out the state transitions or expected rewards:

$$p(s'|s,a) = \sum_r p(s', r|s, a),$$

to obtain the probability of arriving to a certain state.

Also, the expected reward:

$$\mathbb{E}[R|s,a] = \sum_r r \sum_{s'} p(r, s'|s, a).$$

There is an alternative equivalent definition, which introduces the notion of the expected reward into the concept, and takes it out of the probabity function:

Definition 7.2. A MDP is a tuple (S, A, p, r, γ) where:

- S is the set of all possible states with the Markov Property.
- A is the set of all possible actions.
- $p(s'|s,a)$ is the probability of transitioning to s', fiven a state s and an action a.
- $r : S \times A \to \mathbb{R}$ is the expected reward, achieved on a transition starting in (s,a),

$$r = \mathbb{E}[R|s,a].$$

- $\gamma \in [0,1]$ is a discount factor that trades off later rewards to earlier ones.

Now, we have to clarify what is the Markov Property:

Definition 7.3. Consider a sequence of random variables, $\{S_t\}_{t\in\mathbb{N}}$, indexed by time and taken from a set of states S. Consider also the set of actions A and rewards in $\mathbb{R}$.
A state s has the **Markov Property** when, for all $s' \in S$,

$$p(S_{t+1} = s'|S_t = s) = p(S_{t+1} = s'|h_{t-1}, S_t = s),$$

for all possible histories $h_{t-1} = \{S_1, ..., S_{t-1}, A_1, ..., A_{t-1}, R_1, ..., R_{t-1}\}$.

Therefore, in an MDP, the current state captures all relevant information from the history, it is a sufficient statistic of the past. So, once the state is known, the history may be thrown away.

Exercise 7.1. In an MDP, which of the following statemest are true?

1. $p(S_{t+1} = s'|S_t = s, A_t = a) = p(S_{t+1} = s'|S_1, ..., S_{t-1}, A_1, ..., A_t, S_t = s)$: false, the RHS does not condition on $A_t = a$.
2. $p(S_{t+1} = s'|S_t = s, A_t = a) = p(S_{t+1} = s'|S_1, ..., S_{t-1}, S_t = s, A_t = a)$: true.
3. $p(S_{t+1} = s'|S_t = s, A_t = a) = p(S_{t+1} = s'|S_1, ..., S_{t-1}, S_t = s)$: false, the RHS does not condition on $A_t = a$.
4. $p(R_{t+1} = r, S_{t+1} = s'|S_t = s) = p(R_{t+1} = r, S_{t+1} = s'|S_1, ..., S_{t-1}, S_t = s)$: true.

It is also worth noting that most MDPs are discounted, and there are several reasons for these:

- Problem specification: immediate rewards may actually be more valuable. For instance, animal/human behavior shows preference for immediate reward.

- Solution side: it is mathematically convenient to discount rewards, because it allows for easier proofs of convergence, and avoids infinite returns in cyclic Markov processes.

As we outlined previously, the **goal of an RL agent** is to find a behavior policy that maximises the expected return G_t. Recall our definition for value funtion

$$v_\pi(s) = \mathbb{E}[G_t | S_t = s, \pi].$$

Similarly, we can define the **state-action values**, as

$$q_\pi(s,a) = \mathbb{E}[G_t | S_t = s, A_t = a, \pi],$$

and there is the following connection between them:

$$v_\pi(s) = \sum_a \pi(a|s)\, q_\pi(s,a) = \mathbb{E}[q_\pi(S_t, A_t) | S_t = s, \pi], \forall s \in S.$$

Also, we can define the maximum possible value functions:

Definition 7.4. The **optimal state value function**, $v^*(s)$, is the maximum value function over all policies,

$$v^*(s) = \max_\pi v_\pi(s).$$

The **optimal state-action value function**, $q^*(s,a)$, is the amximum state-action value function over all policies,

$$q^*(s,a) = \max_\pi q_\pi(s,a).$$

The optimal value function specifies the best possible performance in the MDP. We can consider the MDP to be solved when we know the optimal value function.

In addition, value functions allow us to define a partial ordering over policies, having

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s), \forall s \in S.$$

With this partial ordering, the following theorem state that optimal policies exist for every MDP:

Theorem 7.1. ***Optimal Policies Theorem***
For any MDP:

- *There exists an optimal policy π^* that is better than or equal to all other policies,*

$$\pi^* \geq \pi, \forall \pi.$$

- *All optimal policies achieve the optimal value function,*

$$v^{\pi^*}(s) = v^*(s).$$

- *All optimal policies achieve the optimal state-action value function,*

$$q^{\pi^*}(s,a) = q^*(s,a).$$

To find an optimal policy, we can maximise over $q^*(s,a)$:

$$\pi^*(s,a) = \begin{cases} 1 & if\ a = \arg\max_{a \in A} q^*(s,a) \\ 0 & otherwise \end{cases}.$$

That is, the optimal policy is to take action a in state s if a is the action that gives the highes state-action value given state s.

Remark 7.2. There is always a deterministic optimal policy for any MDP, and we know $q^*(s,a)$, we know the optimal policy immediately.

Also, there can be multiple optimal policies, and if multiple actions maximize $q_*(s,\cdot)$, we can pick any of them.

Now, recall the Bellman Equations we saw previously. The following theorem explains how to express the value functions by means of these equations:

Theorem 7.2. ***Bellman Expectation Equations***
Given an MDP, $M = (S, A, p, r, \gamma)$, for any policy π, the value functions obey the followin expectation equations:

$$v_\pi(s) = \sum_a \pi(s,a)\left[r(s,a) + \gamma \sum_{s'} p(s'|a,s)\, v_\pi(s')\right],$$

and

$$q_\pi(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|a,s) \sum_{a' \in A} \pi(a'|s')\, q_\pi(s',a').$$

Theorem 7.3. ***Bellman Optimality Equations***
Given an MDP, $M = (S, A, p, r, \gamma)$, the optimal value functions obey the following expectation equations:

$$v^*(s) = \max_{a \in A}\left[r(s,a) + \gamma \sum_{s'} p(s'|a,s)\, v^*(s')\right],$$

$$q^*(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|a,s) \max_{a' \in A} q^*(s',a').$$

Remark 7.3. There can not be a policy with a higher value than $v^*(s) = \max_\pi v_\pi(s), \forall s$.

Intuition on the proof for the Bellman Optimality Equations*:*

An optimal policy can be found by maximising over $q^*(s,a)$,

$$\pi^*(s,a) = \begin{cases} 1 & if\ a = \arg\max_{a \in A} q^*(s,a), \\ 0 & otherwise. \end{cases}$$

Applyin the Bellman Expectation Equation:

$$\begin{aligned} q_{\pi^*}(s,a) =& r(s,a) + \gamma \sum_{s'} p(s'|a,s) \sum_{a' \in A} \pi^*(a'|s')\, q_{\pi^*}(s',a') \\ =& r(s,a) + \gamma \sum_{s'} p(s'|a,s) \max_{a' \in A} q^*(s',a'). \end{aligned}$$

We can also express the Bellman equations in matrix form, as

$$V = R^\pi + \gamma P^\pi V,$$

where $v_i = v(s_i)$, $R_i^\pi = \mathbb{E}[R_{t+1}|S_t = s_i, A_t \sim \pi(S_t)]$ and $P_{ij}^\pi = p(s_j|s_i) = \sum_a \pi(a|s_i)\, p(s_j|s_i,a)$.
This is a linear equations, that can be solved directly:

$$\begin{aligned} (I - \gamma P^\pi) V =& R^\pi \\ V =& (I - \gamma P^\pi)^{-1} R^\pi. \end{aligned}$$

The computational complexity is $O\left(|S|^3\right)$, making this only feasible for small problems. This makes it helpful to design other methods for larger problems. For example, there are iterative methods such as dynamic programming, monte-calro evaluation and temporal-difference learning.

7.1 Solving Reinforcement Learning Problems with Bellman Equations

Pb1. Estimating v_π or q_π is called **policy evaluation**, or **prediction**:

(a) Given a policy, what is my expected return under that behavior?

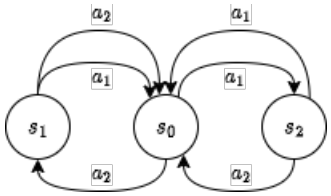
(b) Given this treatment protocol/trading strategy, what is my expected return?

Pb2. Estimating v_* or q_* is sometimes called **control**, because these can be used for **policy optimisation**:

(a) What is the optimal way of behaving? What is the optimal value function?

(b) What is the optimal treatment? What is the optimal control policy to minimise time, fuel consumption, etc?

Exercise 7.2. Consider the following MDP:



The actions have a 0.9 probability of success and with 0.1 probability we remain in the same state.

$R_t = 0$ for all transitions that end up in S_0 and $R_t = -1$ for all other transitions.

Discount factor: $\gamma = 0.9$.

Questions:

- What is v_π for $\pi(s) = a_1(\rightarrow), \forall s$?
  According to the Bellman Equations:

$$v_\pi(s) = \sum_a \pi(s,a)\left[r(s,a) + \gamma \sum_{s'} p(s'|a,s)\, v_\pi(s')\right],$$

$$\begin{aligned} v_\pi(s_0) =& \pi(s_0, a_1\checkmark)\left[r(s_0, a_1\checkmark) + 0.9 \cdot \left[p(s_1|a_1,s_0)\, v_\pi(s_1) + p(s_2|a_1,s_0)\, v_\pi(s_2)\right]\right] + \pi(s_0, a_1\times) \cdot \left[r(s_0, a_1\times) + 0.9 \cdot p(s_0\right. \\ =& 0.9 \cdot \left[-1 + 0.9 \cdot \left[0 + v_\pi(s_2)\right]\right] + 0.09 \cdot v_\pi(s_0) \\ =& -0.9 + 0.81 \cdot v_\pi(s_2) + 0.09 \cdot v_\pi(s_0). \end{aligned}$$

Isolating, $v_\pi(s_0)$, we get

$$0.91 v_\pi(s_0) = -0.9 + 0.81 \cdot v_\pi(s_2) \implies v_\pi(s_0) = -0.99 + 0.89 \cdot v_\pi(s_2).$$

Now, let's go for $v_\pi(s_2)$:

$$\begin{aligned} v_\pi(s_2) =& \pi(s_2, a_1\checkmark)\left[r(s_2, a_1\checkmark) + 0.9 \cdot \left[p(s_0|a_1,s_2)\, v_\pi(s_0) + p(s_1|a_1,s_2)\, v_\pi(s_1)\right]\right] + \pi(s_2, a_1\times) \cdot \left[-1 + 0.9 \cdot v_\pi(s_2)\right] \\ =& 0.9 \cdot \left[0 + 0.9 \cdot \left[v_\pi(s_0) + 0\right]\right] + 0.1\left[-1 + 0.9 \cdot v_\pi(s_2)\right] \\ =& 0.81 \cdot v_\pi(s_0) - 0.1 + 0.09 \cdot v_\pi(s_2) \\ =& 0.81 \cdot \left(-1 + 0.9 \cdot v_\pi(s_2)\right) - 0.1 + 0.09 \cdot v_\pi(s_2) \\ =& -0.91 + 0.82 \cdot v_\pi(s_2). \end{aligned}$$

Therefore,

$$v_\pi(s_2) = \frac{-0.91}{0.18} = -5.06.$$

This means that

$$v_\pi(s_0) = -5.49.$$

Finally,

$$\begin{aligned} v_\pi(s_1) =& \pi(s_1, a_1)\left[r(s_1, a_1) + 0.9 \cdot \left[p(s_0|a_1, s_1)\, v_\pi(s_0) + p(s_2|a_1, s_1)\, v_\pi(s_2)\right]\right] + \pi(s_1, a_1\times) \cdot \left[-1 + 0.9 \cdot v_\pi(s_1)\right] \\ =& 0.9 \cdot [0 + 0.9 \cdot [-5.49 + 0]] + 0.1 \cdot [-1 + 0.9 \cdot v_\pi(s_1)] \\ =& -4.45 - 0.1 + 0.09 \cdot v_\pi(s_1). \\ =& -4.55 + 0.09 \cdot v_\pi(s_1). \end{aligned}$$

So

$$v_\pi(s_1) = \frac{-4.55}{0.91} = -5.$$

That is, $v_\pi(s_0) = -5.49, v_\pi(s_1) = -5$ and $v_\pi(s_2) = -5.06$.

- What is v_π for the uniformly random policy?

  This is: with probability 0.1, stay in the same state, with probability 0.45 choose a_1 and with probability 0.45 choose a_2. For s_0:

$$\begin{aligned} v_\pi(s_0) =& 0.1 \cdot 0.9 \cdot v_\pi(s_0) + 0.45 \cdot [-1 + 0.9 \cdot v_\pi(s_2)] + 0.45 \cdot [-1 + 0.9 \cdot v_\pi(s_1)] \\ =& 0.09 \cdot v_\pi(s_0) - 0.9 + 0.405 \cdot v_\pi(s_2) + 0.405 \cdot v_\pi(s_1), \end{aligned}$$

  so

$$v_\pi(s_0) = -0.99 + 0.445 \cdot v_\pi(s_2) + 0.445 \cdot v_\pi(s_1).$$

  For s_1:

$$\begin{aligned} v_\pi(s_1) =& 0.1 \cdot [-1 + 0.9 \cdot v_\pi(s_1)] + 0.45 \cdot [0 + 0.9 \cdot v_\pi(s_0)] + 0.45 \cdot [0 + 0.9 \cdot v_\pi(s_0)] \\ =& -0.1 + 0.09 \cdot v_\pi(s_1) + 0.81 \cdot v_\pi(s_0), \end{aligned}$$

  so

$$v_\pi(s_1) = -0.1 + 0.9 \cdot v_\pi(s_0).$$

  For s_2:

$$\begin{aligned} v_\pi(s_2) =& 0.1 \cdot [-1 + 0.9 \cdot v_\pi(s_2)] + 0.45 \cdot [0 + 0.9 \cdot v_\pi(s_0)] + 0.45 \cdot [0 + 0.9 \cdot v_\pi(s_0)] \\ =& -0.1 + 0.09 \cdot v_\pi(s_2) + 0.81 \cdot v_\pi(s_0), \end{aligned}$$

  so

$$v_\pi(s_2) = -0.1 + 0.9 \cdot v_\pi(s_0).$$

  Therefore_

$$\begin{aligned} v_\pi(s_0) =& -0.99 + 0.89 \cdot (-0.1 + 0.9 \cdot v_\pi(s_0)) \\ =& -0.99 - 0.089 + 0.8 \cdot v_\pi(s_0) \\ =& -1.08 + 0.8 \cdot v_\pi(s_0). \end{aligned}$$

  That is,

$$v_\pi(s_0) = -5.4.$$

  And,

$$v_\pi(s_1) = v_\pi(s_2) = -4.96.$$

- Same policy evaluation problems for $\gamma = 0$? What do you notice?

  First, $\pi(s) = a_1(\rightarrow), \forall s$:

$$\begin{aligned} v_\pi(s_0) =& 0.1 \cdot 0 + 0.9 \cdot (-1) \\ =& -0.9. \end{aligned}$$

$$\begin{aligned} v_\pi(s_1) =& 0.1 \cdot (-1) + 0.9 \cdot (0) \\ =& -0.1. \end{aligned}$$

$$v_\pi(s_2) = -0.1.$$

Second, $\pi(s)$ the random policy:

$$\begin{aligned} v_\pi(s_0) =& 0.1 \cdot 0 + 0.45 \cdot (-1) + 0.45 \cdot (-1) \\ =& -0.9. \end{aligned}$$

$$\begin{aligned} v_\pi(s_1) =& 0.1 \cdot (-1) + 0.9 \cdot (0) \\ =& -0.1. \end{aligned}$$

$$v_\pi(s_2) = -0.1.$$

We can observe that if we don't take the discount factor into account, two very different policies can give us the same (short term) values.

7.2 Dynamic Programming

> Dynamic programming refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov Decision Process (MDP).
>
> -Sutton & Barto, 2018

We will discuss several dynamic programming methods to solve MDPs, all of which consist of two important parts:

- Policy evaluation.
- Policy improvement.

7.2.1 Policy Evaluation

We start by discussing how to estimate

$$v_\pi(s) = \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) \,|s, \pi\right].$$

The idea is to turn the equality into an update rule. The process is described in the following algorithm:

```
def policy_eval(S, R, pi, gamma):

  v = [0 for s in S]

  repeat until converge:
    for s in S:
      v_new(s) = E[R + gamma * v(S_t+1) | S_t = s, pi]

    v = v_new

  return v
```

Note that this algorithm always converge under appropriate conditions, like $\gamma < 1$. We will delve into this later.

Example 7.1. Policy Evaluation example.

Take the following MDP:

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

The possible actions are to go up, down, left, right and when you reach the red cells, you finish. Each transitions costs -1 point.

Let's evaluate the random policy with $\gamma = 1$.

Initialization:

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Step 1: we do

$$v(s) = \mathbb{E}[R + \gamma v(S) | S, \pi],$$

so for example, for cell (1), it is:

$$v(1) = \frac{1}{3}(-1 + 1 \cdot 0) \cdot 3 = -1.$$

0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	0

Step 2:

$$v(1) = \frac{1}{3}(-1 + 1 \cdot 0) + \frac{2}{3}(-1 - 1) = -1.7.$$

0	-1.7	-2	-2
-1.7	-2	-2	-2
-2	-2	-2	-1.7
-2	-2	-1.7	0

Step 3:

$$v(1) = \frac{1}{3}(-1 + 0) + \frac{2}{3}(-1 - 2) = -2.3.$$

0	-2.3	-2.9	-3
-2.3	-2.9	-3	-2.9
-2.9	–3	-2.9	-2.3
-3	-2.9	-2.3	0

And so on... It would converge at

0	-14	-20	-22
-14	-18	-20	-20
-20	-20	-18	-14
-22	-20	-14	0

7.2.2 Policy Improvement

We can use the values computed with policy evaluation to improve the policy. The simplest way to achieve this is with a **greedy policy improvement** approach, which is as follows:

```
def policy_improvement(S, R, pi, gamma)
  pi_new = {}

  v = [0 for s in S]

  for s in S:
    v(s) = policy_eval(S, R, pi, gamma)
    pi_new[v] = argmax_a E[R + gamma*v(S_t+1) | S_t = s, A_t = a]
    pi = pi_new

  return pi_new
```

Claim 7.1. It is possible to show that

$$v_{\pi_{new}}(s) \geq v_{\pi}(s), \forall s.$$

Example 7.2. We can use this greedy approach combined with the previous example:

Initialization, with random policy:

0	0	0	0	0	$\downarrow, \leftarrow, \rightarrow$	$\downarrow, \leftarrow, \rightarrow$	$\downarrow, \leftarrow$
0	0	0	0	$\uparrow, \downarrow, \rightarrow$	$\uparrow, \downarrow, \leftarrow, \rightarrow$	$\uparrow, \downarrow, \leftarrow, \rightarrow$	$\uparrow, \downarrow, \leftarrow$
0	0	0	0	$\uparrow, \downarrow, \rightarrow$	$\uparrow, \downarrow, \leftarrow, \rightarrow$	$\uparrow, \downarrow, \leftarrow, \rightarrow$	$\uparrow, \downarrow, \leftarrow$
0	0	0	0	$\uparrow, \rightarrow$	$\uparrow, \leftarrow, \rightarrow$	$\uparrow, \leftarrow, \rightarrow$	

Step 1:

0	-1	-1	-1	0	$\leftarrow$	$\downarrow, \leftarrow, \rightarrow$	$\downarrow, \leftarrow$
-1	-1	-1	-1	$\uparrow$	$\uparrow, \downarrow, \leftarrow, \rightarrow$	$\uparrow, \downarrow, \leftarrow, \rightarrow$	$\uparrow, \downarrow, \leftarrow$
-1	-1	-1	-1	$\uparrow, \downarrow, \rightarrow$	$\uparrow, \downarrow, \leftarrow, \rightarrow$	$\uparrow, \downarrow, \leftarrow, \rightarrow$	$\downarrow$
-1	-1	-1	0	$\uparrow, \rightarrow$	$\uparrow, \leftarrow, \rightarrow$	$\rightarrow$	

Step 2:

0	-1.7	-2	-2	0	$\leftarrow$	$\leftarrow$	$\downarrow, \leftarrow$
-1.7	-2	-2	-2	$\uparrow$	$\uparrow, \leftarrow$	$\uparrow, \downarrow, \leftarrow, \rightarrow$	$\downarrow$
-2	-2	-2	-1.7	$\uparrow$	$\uparrow, \downarrow, \leftarrow, \rightarrow$	$\downarrow, \rightarrow$	$\downarrow$
-2	-2	-1.7	0	$\uparrow, \rightarrow$	$\rightarrow$	$\rightarrow$	

Step 3:

0	-2.3	-2.9	-3	0	$\leftarrow$	$\leftarrow$	$\downarrow, \leftarrow$
-2.3	-2.9	-3	-2.9	$\uparrow$	$\uparrow, \leftarrow$	$\uparrow, \downarrow, \leftarrow, \rightarrow$	$\downarrow$
-2.9	–3	-2.9	-2.3	$\uparrow$	$\uparrow, \downarrow, \leftarrow, \rightarrow$	$\downarrow, \rightarrow$	$\downarrow$
-3	-2.9	-2.3	0	$\uparrow, \rightarrow$	$\rightarrow$	$\rightarrow$	

Step converged:

0	-14	-20	-22	0	$\leftarrow$	$\leftarrow$	$\downarrow, \leftarrow$
-14	-18	-20	-20	$\uparrow$	$\uparrow, \leftarrow$	$\uparrow, \downarrow, \leftarrow, \rightarrow$	$\downarrow$
-20	-20	-18	-14	$\uparrow$	$\uparrow, \downarrow, \leftarrow, \rightarrow$	$\downarrow, \rightarrow$	$\downarrow$
-22	-20	-14	0	$\uparrow, \rightarrow$	$\rightarrow$	$\rightarrow$	

Observe how in the second iteration we already found the optimal policy!

In this example, we showed how we can use evaluation to improve our policy, and in fact we obtained the optimal policy. However, the greedy approach does not always ensure reaching the optimal policy.

This approach is called **policy iteration**:

- Policy evaluation: estimate v^{π}.
- Policy improvement: generate $\pi' \geq \pi$.

It is natural to ask if policy evaluation need to converge to v^{π} or if we should stop the evaluation at some point. Ways to stop it are to put a threshold of minimum change between iterations, or simply after k iterations. One extreme, which is in fact quite usual in practice, is to stop after $k = 1$, which is equivalent to **value iteration**:

```
def value_iter(S, R, pi, gamma):

  v = [0 for s in S]

  repeat until converge
    for s in S:
      v_new(s) = max_a E[R + gamma * v(S_t+1) | S_t = s, A_t = a]

    v = v_new

  return v
```

In the following table, we sum up the different approaches:

Problem	Bellman Equation	Algorithm
Prediction	Bellman Expectation Eq	Iterative Policy Eval
Control	Bellman Expectation Eq + (Greedy) Policy Improvement	Policy Iteration
Control	Bellman Optimality Eq	Value Iteration

Observations

- The algorithms are based on state-value functions $v_{\pi}(s)$ or $v^{*}(s)$, with complexity $O\left(|A|\,|S|^2\right)$ per iteration.
- It could also be applied to action-value functions $q_{\pi}(s,a)$ or $q^{*}(s,a)$, with complexity $O\left(|A|^2\,|S|^2\right)$ per iteration.

7.3 Extensions to Dynamic Programming

7.3.1 Asynchronous Dynamic Programming

DP methods described so far used **synchronous** updates, meaning all states are updated in parallel. In contrast, **asynchronous DP** backs up states individually, in any order. This can significantly reduce computation, and it is guaranteed to converge if all states continue to be selected.

We are going to see three approaches for ADP:

In-Place DP Before, with synchronous value iteration, we stored two copies of the value function:

$$v_{new}(s) = \max_a \mathbb{E}\left[R_{t+1} + \gamma v\left(S_{t+1}\right) | S_t = s, A_t = a\right],$$

$$v = v_{new}.$$

Now, in-place value iteration stores only one copy of the value function, doing:

$$v(s) = \max_a \mathbb{E}\left[R_{t+1} + \gamma v\left(S_{t+1}\right) | S_t = s, A_t = a\right].$$

Prioritised Sweeping We can use the magnitude of the Bellman error to guide the state selection. For example:

$$E = \left| \max_a \mathbb{E}\left[R_{t+1} + \gamma v\left(S_{t+1}\right) | S_t = s, A_t = a\right] - v\left(s\right) \right|.$$

We then backup the state with the largest remaining Bellman error, and update the Bellman error of affected states after each backup. This reuires knowledge of reverse dynamics (which are the predecessor states). It can be implemented efficiently with a priority queue.

Real-Time DP The idea in this case is to only update states that are relevant to the agent. For example, if the agent is in state S_t, we update that state value, or the states that it expects to be in soon.

Full-Width backups Standard DP uses full-width backups. This means that, for each backup, it being synchronous or asynchronous:

- Every successor state and action is considered.
- Using the true model of transitions and reward function.

DP is effective for medium-sized problems (with millions of states). For large problems DP suffers from the curse of dimensionality, since the number of states grows exponentially with the number of state variables, and even one full backup can be too expensive.

Sample Backups This approach consists in using sample rewards and sample transitions (s, a, r, s') instead of the reward function R, and the transition dynamics, P.

It presents some advantages:

- It's model free: knowledge about the MDP is not required.
- Breaks the curse of dimensionality through sampling.
- Cost of backup is constant, independent of $n = |S|$.

References

[1] Tom Dupuis. Machine learning. Lecture Notes.