# 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

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

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 $L^\infty$, 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.
  - **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, $\theta$, which are learnt during training.

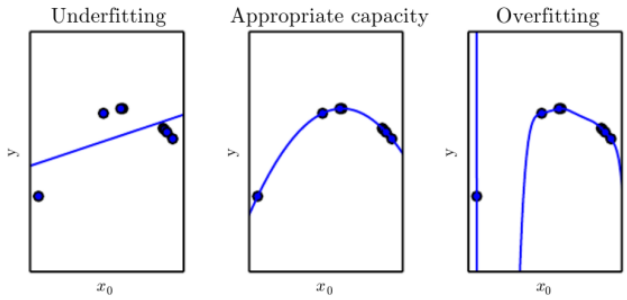



Figure 1: Appropriate capacity, overfitting and underfitting visualization.

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 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 $\theta$, 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 $\theta$, $p(x;\theta)$ 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:

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

# 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:

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

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

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

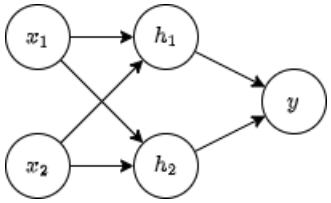
This function preserves much of the good oprimization properties of a linear function, i.e., it is differentiable (apart from one point), and its derivative is constant.

**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:

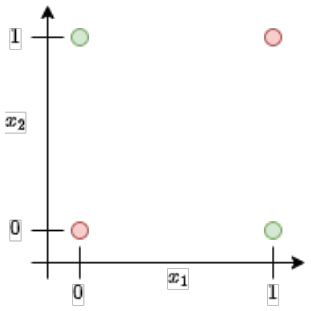
$$W_h = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, b_h = \begin{pmatrix} 0 \\ -1 \end{pmatrix}, W_y = \begin{pmatrix} 1 \\ -2 \end{pmatrix}, b = 0.$$

Let's check:

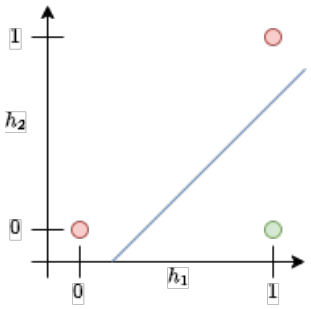
| $x_1$ | $x_2$ | $h$ | $y$ |
|---|---|---|---|
| 0 | 0 | $\begin{pmatrix} 0 \\ -1 \end{pmatrix}$ | $\begin{pmatrix} 1 & -2 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \end{pmatrix} = 0 \leq 0$ |
| 0 | 1 | $\begin{pmatrix} 1 \\ 1-1 \end{pmatrix}$ | $\begin{pmatrix} 1 & -2 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = 1 > 0$ |
| 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 $p_{model}(x=i)$ with to-be-optimized parameters $\theta$ and let the frequency of outcome $i$ in the training set be $p(x=i)$. Given $N$ conditionally independent samples in the training set, then the likelihood of the parameters $\theta$ of the model $p_{model}(x=i)$ on the training set is:

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

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

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

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

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

*as an approximate realization of the function $f$, that is*

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

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

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

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 $\varepsilon$ 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

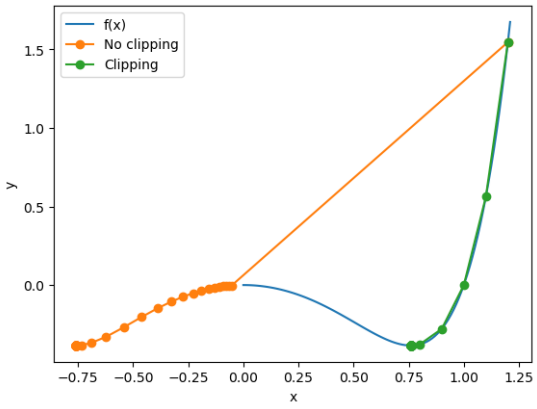
It's important to stabilize and improve the convergence of gradient descent on DNNs, and for this we need good optimization techniques.

**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

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

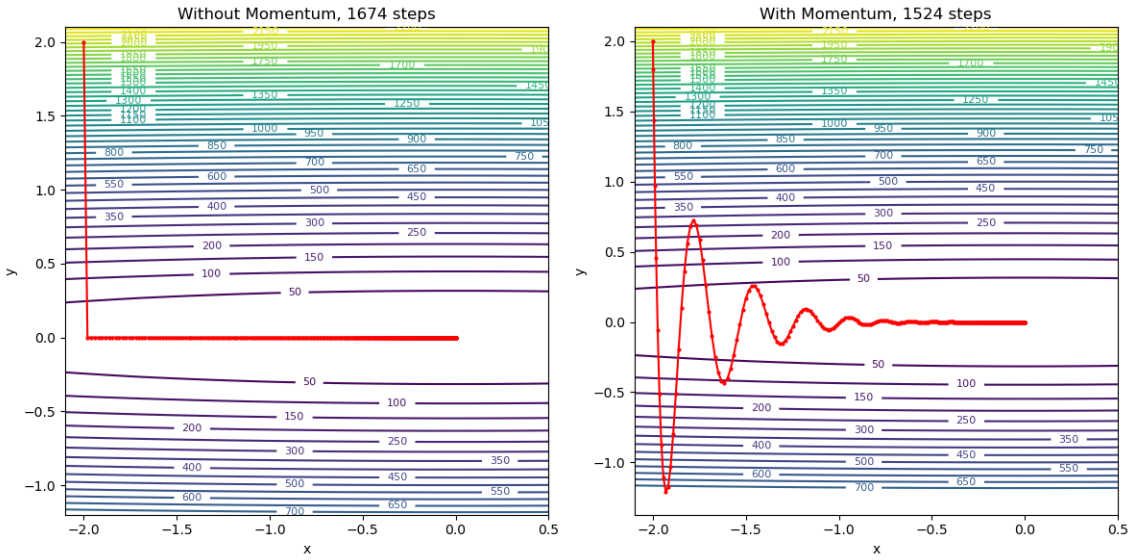
where $0 \leq \alpha < 1$ is a parameter controlling how much of the gradient we use for the parameter update, usually set as 0.9. If it is set to 0, we obtain the vanilla SGD.

The updates are done as

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

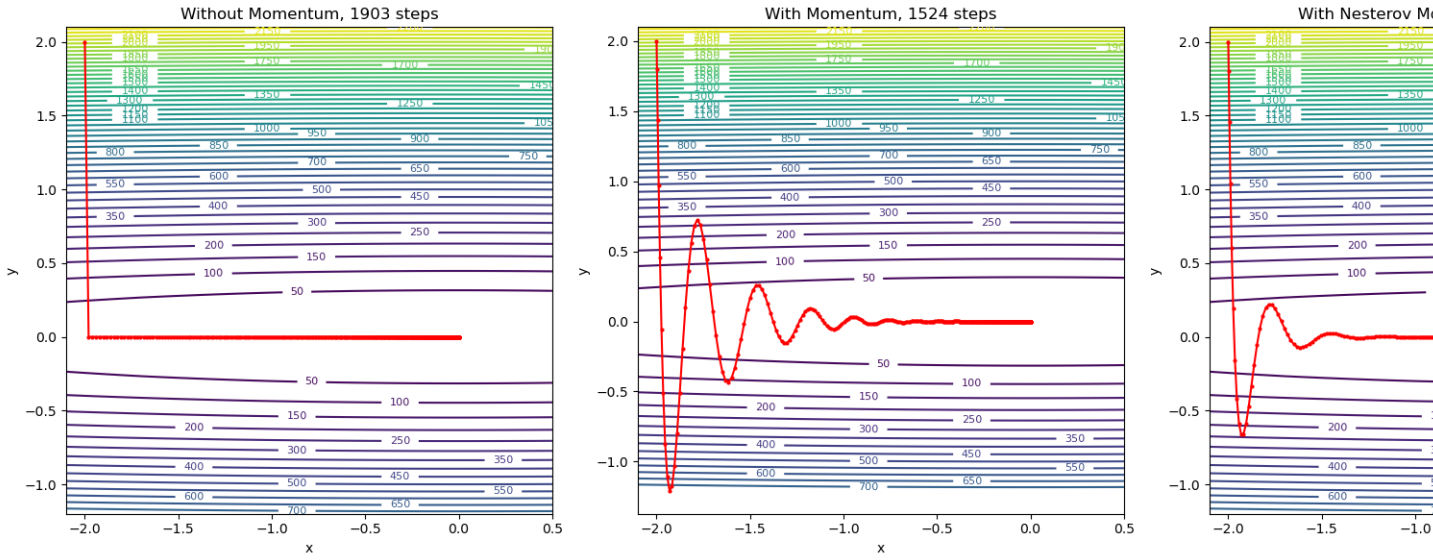
where $\eta$ is the learning rate.

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 $\theta$ 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:

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

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

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

and $\eta_\tau$ after $\tau$ iterations.

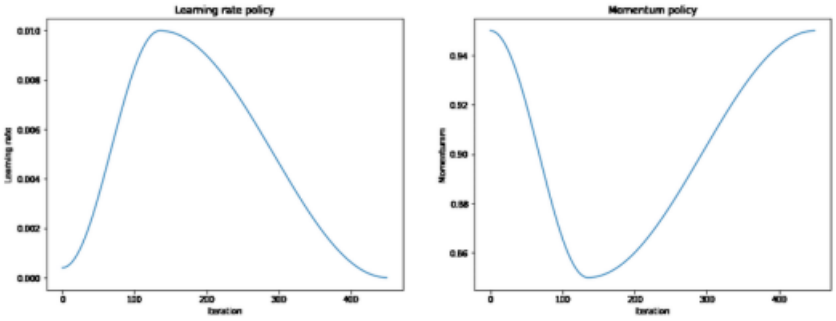
Now, the question is, how to choose $\eta_0, \eta_\tau$ and $\tau$? There are several options:

- With trail and error (using train/validation sets).
- It's better to monitor the loss function during training.
- A practical choice:
  - Choose $\tau$ so that the whole training dataset is seen around 100 times.
  - Choose $\eta_\tau$ around 1% of $\eta_0$.
  - $\eta_0$ comes with experience:
    - Too big: big variations in loss
    - Too small: learning is slow, we can get stuck in a plateau
  - Recipe:
    - Try multiple $\eta_0$ over 100 iterations
    - Pick $\eta_0$ slightly higher than the best

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

**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 $\varepsilon$ 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 $\varepsilon$ 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 $\lambda$ is the **regularization parameter**, controlling the strength of the regularization:

- If $\lambda$ is small, there is only a small regularization, allowing higher weights.
- If $\lambda$ 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)

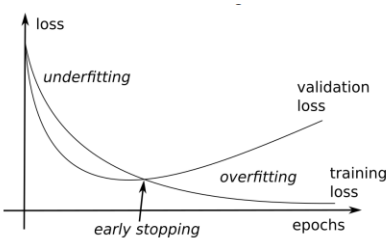
In this case, we penalize the absolute value of the weights, instead of their euclidean norm:

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

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 $\theta$, 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.

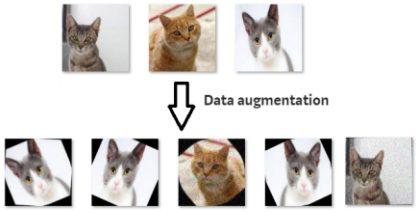
The probability of dropout, $p$, is a hyperparameter. Therefore, what we do at training time is, at each step, deactivate some neurons, randomly. This incentivizes generalization, since relying on some neurons specializing a lot for some features of the training data is harder if these neurons are not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



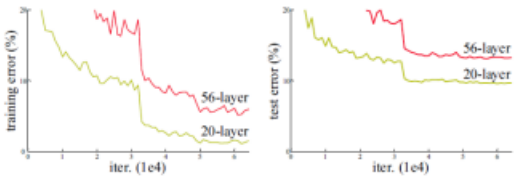
**Mixup** is a particular case of data augmentation, consisting on creating new training new samples with labels by interpolation:

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

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.

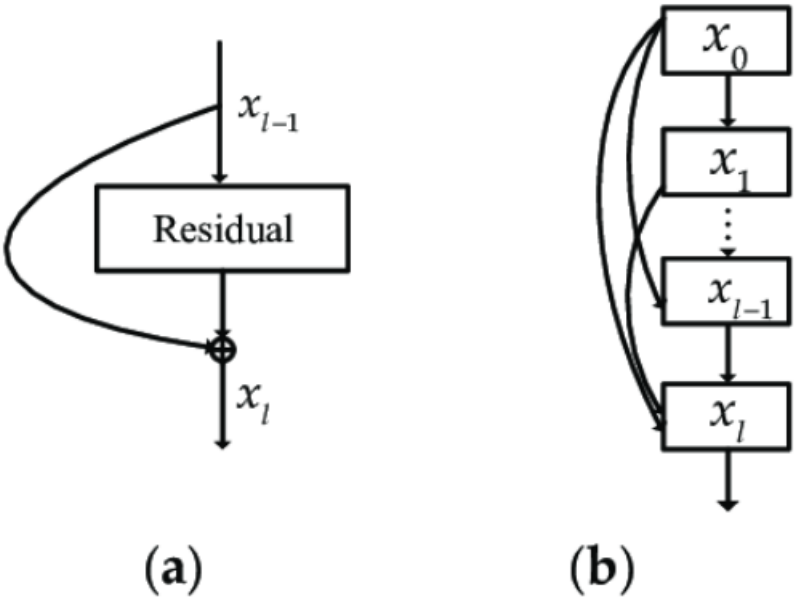
We saw that exploding gradients can be solved by gradient clipping. But vanishing gradients are still the current limitation of deep NN. The solutions include the utilization of ReLU activaiton functions, unsupervised pre-training, batch normalization, residual networks, etc.

### 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

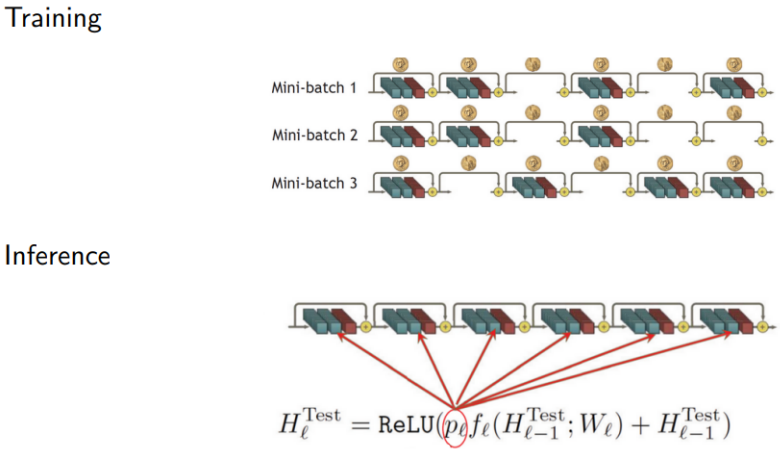
Stochastic depth is a training technique for deep neural networks, particularly effective for very deep networks like ResNets. It was introduced as a solution to the problem of vanishing gradients and long training times in deep networks.

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.

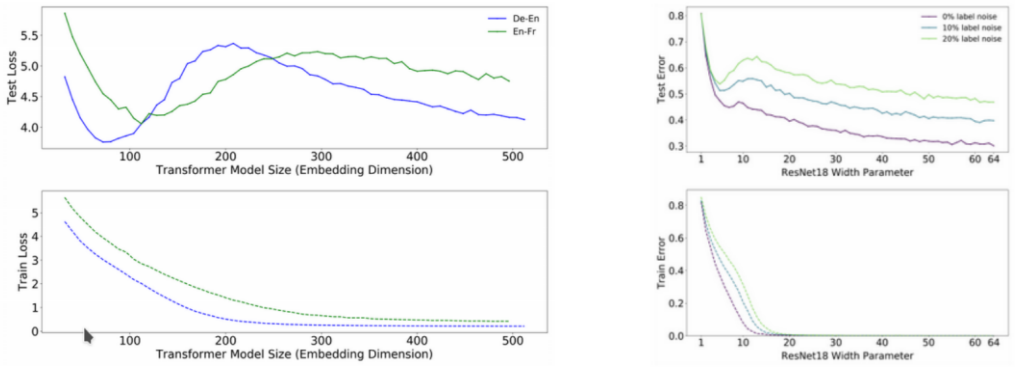
It shows the following phases:

- **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.

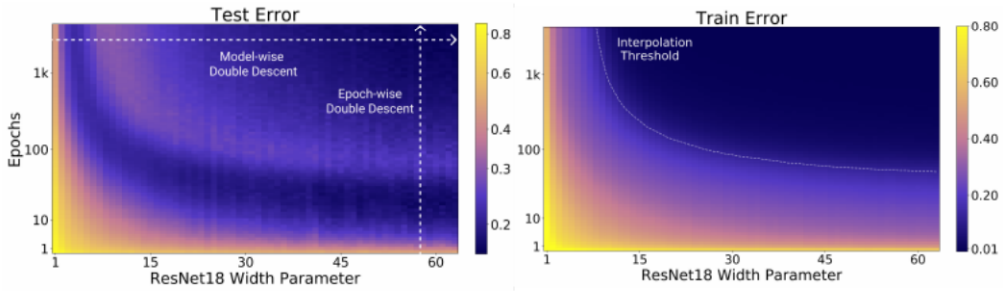
The double descent phenomenon is especially noticeable in scenarios with limited training data. With more data, the peak of the curve (at the interpolation threshold) becomes less pronounced.

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!




# 5. Convolutional Neural Networks

## 5.1. Introduction

When working with MLP, we find several limitations:

1. Dealing with data with a high number of features. For example, let imagine we are working with a dataset of images at full HD resolution. The feature size is $1920 \times 1080 \times 3 = 6\ 220\ 800$. Learning a single layer to reduce the dimension to 1000 requires around $6000K \times 1K \sim 10^9$ parameters, making the training really difficult.
2. Use the knowledge of the input data modality to shape the network. E.g., for images, the network should be equivariant to translation. This means that a pattern should be detected independently of where it is located in the image.

**Convolutional Neural Networks (CNNs)** appear to mitigate these limitations.

> **Definition 5.1.** A CNN is an NN that uses convolutions in place of general matrix multiplication in, at least, one of their layers.

CNNs deal with data arranged according to a grid, which can be temporal date, images, videos, etc.

### 5.1.1. History of CNNs

- The first work on CNNs was done in 1998, by LeCun et al.
- Theoretical advances to make DNN converge by Hinton et al. in 2006.
- Access to large datasets such as ImageNet thanks to Deng et al. in 2009.
- Advances in hardware technology to scale learning, with better CPUs and the development of GPUs.
- Win of the 2012 ImageNet challenge with AlexNet, using CNNs, by Krizhevsky et al. in 2012, made them famous.

## 5.2. Convolutions

**Definition 5.2.** Given two functions, $f, g : \mathbb{R} \to \mathbb{R}$, their **convolution** is defined as

$$(f * g)(x) = \int_{-\infty}^{\infty} f(z)\, g(x - z)\, dz.$$

For discrete function, it becomes

$$(f * g)(m, n) = \sum_{i,j=-\infty}^{i,j=\infty} f(i, j)\, g(m - i, n - j),$$

where in this case $f$ and $g$ are bidimensional.
$f$ is called the **input signal**.
$g$ is called the **kernel**, or filter.
$f * g$, the convolution output, is the **feature map**.

*Remark* 5.1. Properties of convolutions:

- Commutativity:
$$(f * g)(x) = (g * f)(x).$$
- Distributivity:
$$(f * (g + h))(x) = (f * g)(x) + (f * h)(x).$$
- Associativity:
$$((f * g) * h)(x) = (f * (g * h))(x).$$

*Remark* 5.2. Observe that convolutions can be understood as taking a moving average of $f$ around each point $x$, with weights provided by $g$. If $\int g dx = 1$, then it would represent a proper moving average.

In practice, however, CNNs do not really use convolutions, but **cross-correlations**, which consist in sliding one signal (or function) over another and measuring the similarity at each position. It's a way to track how much one signal resembles another as you shift one of them over time or space. Their definition is very similar to convolutions, but convolutions invert the kernel function, while cross-correlations do not:

$$(f \circledast g)(m, n) = \sum_{i,j=-\infty}^{i,j=\infty} f(m, n)\, g(m + i, n + i).$$

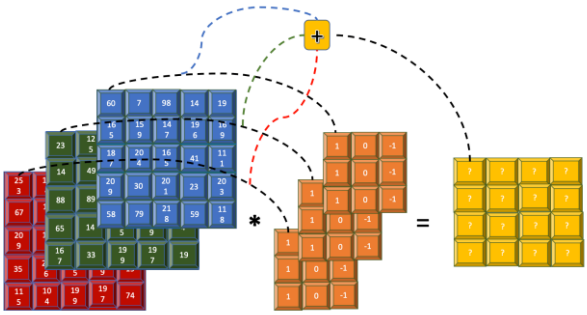


Figure 2: Convolution of a 3-channel input with a 3-channel kernel.

While MLPs make an interaction between each input neurons and each output neurons, CNNs have **sparse interactions**, thanks to the kernels, which have a small number of parameters, significantly reducing the number of parameters of the network.

Moreover, in MLP there is one weight per connection between the input and the output, while CNNs apply the same kernel to different parts of the input, also reducing the number of parameters of the network. This is called the **parameter sharing** property.

**Definition 5.3.** A function $f$ is said to be **equivariant** to a transform $g$, if for any input $x$, it holds

$$f\left(g\left(x\right)\right) = g\left(f\left(x\right)\right).$$

Because of paratemer sharing, the CNNs are equivariant to translations. This makes them particulary useful for some modalities, such as image, for which we aim to detect several objects in the same manner.

Images, and some other modalities, like videos, are composed of **channels**. For example, RGB is a 3-channel way to compose images.

Kernels are convoluted with all channels of the input, as visualized in the following figure.

Convolutions are not equivariant to rotation, zoom, etc. And these might be interesting for our model to be invariant. To make the network invariant to such transformation, we can transform the data with data augmentations including zooming and rotating, and feeding the network with this transformed data.

Notice how convolutions can reduce the input size, whenever the kernel is larger than $1 \times 1$. This is due to the necessity to fit the whole image.

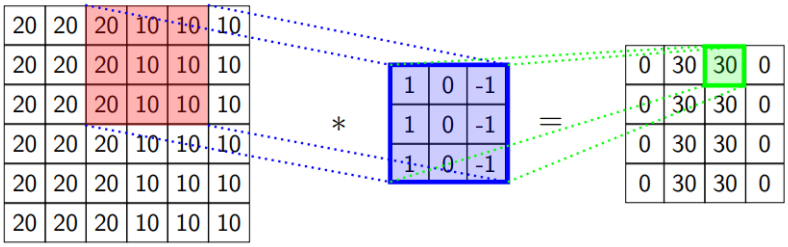
The output size of the convolution of an input of size $n_h \times n_w$ with a kernel of size $k_h \times k_w$ is

$$\left(n - k_h + 1\right) \times \left(n_w - k_w + 1\right).$$

### 5.2.1. Fixed Kernels Weights

If we are interested in certain features, and we know how to define a kernel that is good for identifying these features, we can just do this and applying this kernel.

For example, in the following illustration we observe a kernel that is good at finding vertical edges:




### 5.2.2. Learned Kernel Weights

Learning a CNN backbone using DL is learning the weights of the kernels via backpropagation. The filters seek to learn the best representation to fit its training objective, via the training loss.

Using different kernels allows to detect various patterns, so usually CNNs contain thousands of kernels and are grouped in hierarchical layers, to learn from low-level features to high-level features.

To compute various features, such as different detections (horizontal, vertical, diagonal, etc.) we need several filters. For an input of size $C \times n_1 \times n_2$, where $C$ is the amount of channels, and $F$ filters of size $C \times m_1 \times m_2$, the feature map produced has $F$ channels and is of size

$$F \times (n_1 - m_1 + 1) \times (n_2 - m_2 + 1)\,.$$

### 5.2.3. Receptive fields

The **receptive field** is the region of the input space that a particular CNN's feature is looking at.

### 5.2.4. $1 \times 1$ Convolution

The $1 \times 1$ convolution is a linear combination of all channels for each pixel, allowing to learn to reduce or increase the number of filters.

In general, if the input is of size $F_{in} \times H \times W$ and the filter is of size $F_{out} \times 1 \times 1$, then

- If $F_{out} < F_{in}$: the dimension is reduced.
- If $F_{out} > F_{in}$: the dimension is increased.

$1 \times 1$ convolutions can replace fully connected layers. For example, in classification, we could do $F_{out} = N_{classes}$.

## 5.3. Padding, Stride, Pooling

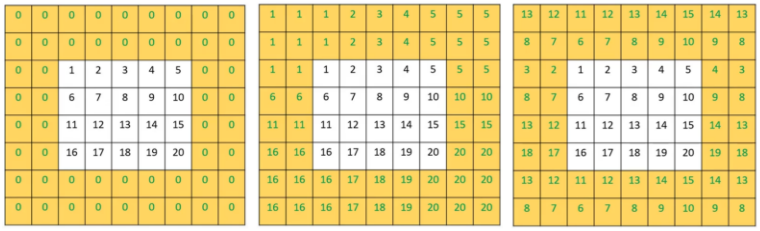
### 5.3.1. The Sides Problem

Convolutions are applied on fixed size patches of the input data, and they can only be shifted until there are no pixels. This introduces two potential problems:

- The output size is reduced.
- The information in the sides of the image can be lost, because the convolution cannot be applied there.

### 5.3.2. Padding

Padding is a solution to the edge problem, consisting on adding data at the sides of the image. There are several strategies, but in general, we just add zeros:



There are several modes of zero padding. For a padding of $k$ pixels:

- Valid: no padding.
- Same: add $\frac{k-1}{2}$ zeros on each side.
- Full: add $k$ zeros on each side.

The output size of a convolution of a $n_h \times n_w$ sized input, to which we add $p_h$ rows and $p_w$ columns of padding, and a kernel of size $k_h \times k_w$ becomes

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1)\,.$$

Generally, we want to have $p_h = k_h - 1$ and $p_w = k_w - 1$, which is achieve by applying same zero padding.

### 5.3.3. Stride

As far as we have seen, we slide the kernel across all locations right and down. However, we could skip some of this locations, effectively reducing the feature map output and the computational cost, but at the expense of the accuracy of the representations.

If we add $s_h, s_w$ for the height and width strides, the feature map size is

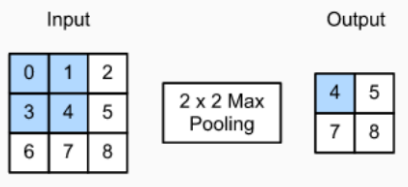
$$\frac{(n_h - k_h + p_h + 1 + s_h)}{s_h} \times \frac{(n_w - k_w + p_w + 1 + s_w)}{s_w}.$$

### 5.3.4. Pooling

Pooling consists in statistically summarizing a neighbourhood. A pooling layer takes a window of values and output one value. There are several pooling types:

- Max Pooling: take the maximum value of a window.
- Average pooling: take the eman value of a window.
- $L_2$ norm, weighted average pooling, etc.

For example, the following illustrates the use of a 2x2 max pooling layer:




Pooling is useful to make our representations invariant to minor translations. For deep layers, minor modifications can mean great changes in the input image.

Also, it reduces the amount of parameters and computation, and help reducing overfiting.

It sets up a strong prior: invariance to local translation of learned kernels.

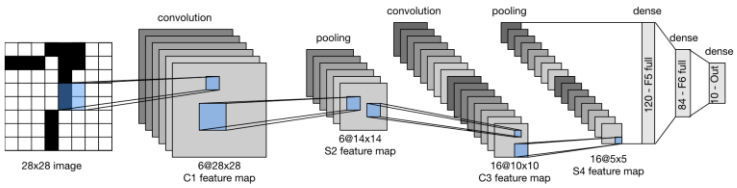
It can be degrading for some tasks, where precise locations are required.

## 5.4. CNNs

**LeNet** by LeCun et al. in 1998 was designed after 10 uears of working with handwritten bank checks. It paved the basics of DL:

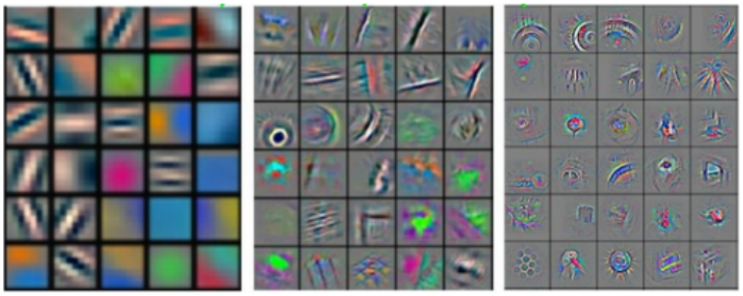
- Feature extraction by convolution layers.
- Classification by MLP layers.
- Layer of convolution followed by average pooling, and non-linearity, with sigmoid or tanh.
- SGD optimizer to perform backpropagation.

The architecture is the following:




**AlexNet** by Krizhevsky et al. in 2012 was on of the first DNN, and won the 2012 ImageNet challenge, causing a new DL area. They extended LeNet to larger images, with wider convolutions, and used ReLUs and max pooling, combined with dropout.

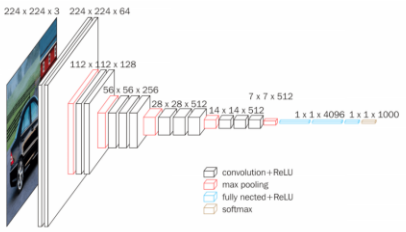
A very interesting fact they observed was that as the layers were deeper, the receptive field was higher: The first layers learn basic patterns, while deeper layers learn more complicated shapes.



**Visual Geometry Group (VGG)** by Simonyan et al. in 2015 was the first network with blocks. the blocks consisted in:

- Convolution layer with padding to keep the same resolution.
- Non-linearity.
- Pooling layer to reduce resolution.

The architecture was the following:




The problem was how to maintain enough resolution to have a lot of blocks. The solution was to introduce more convolutions at low resolution, instead of few convolutions at high resolution.

This increases the number of non-linearities and reduces the number of parameter to achieve an equal receptive field.

**Network in Network (NiN)**, by Lin et al. in 2013: previous architectures improved by enhacing the number of convolutions (width) and deepening the network (depth). This presents two problems:

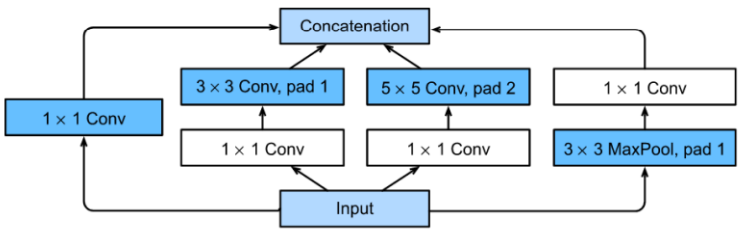
- The last fully connected layers consume large number of parameters.
- It is difficult to add non-linearity without using fully connected layers, that destroy the spatial structure.

NiN made improvements:

- It used $1 \times 1$ convolutiones to increase the number of non-linearities.
- Also, a global average pool at the end to remove large fully connected layers.

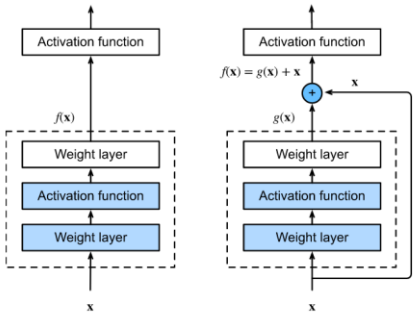
**Inception Blocks**: Google was facing a problem. They needed fast deployment and inference, but large convolutions with lots of channels increase inference time. Their solution was to reduce the number of channels by applying several convolutions in parallel. Instead of choosing a single filter size for a given layer, Inception blocks apply multiple different-sized filters (e.g., 1x1, 3x3, 5x5 convolutions) in parallel to the same input feature map. This allows the network to capture information at various scales.

An inception block looks as follows:



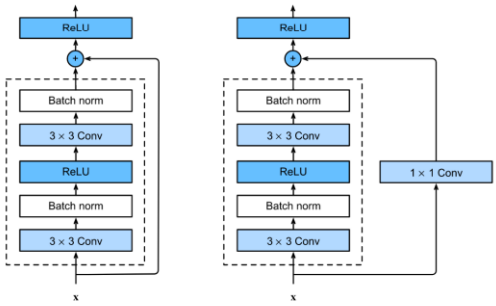
**GoogleNet**, by Szegedy et al. in 2015 was based on inception blocks. It used several classifiers throughout training to enhance discriminative features at first layers, and reduce vanishing gradients.

**Residual Blocks**: these blocks make use of **residual connections**, or shortcut connections, which forward the input to the following layer. That is, if a layer is represented as $f(x)$, the same layer, when used inside a residual block, would be $f(x) + x$. This is illustrated below.

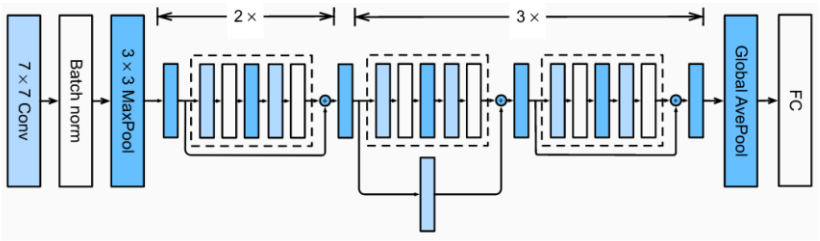



Residual blocks make the identity function easier to learn, and reduce vanishing or exploding gradient problems.

**ResNet Blocks**: these blocks are a combination of VGG blocks and residual blocks. They use $3 \times 3$ convolutions, with batch normalization to stabilize training and residual connection concatenated to the result of applying two of these convolutions. The residual connection can be either direct, or by means of a $1 \times 1$ convolution. This is illustrated below:



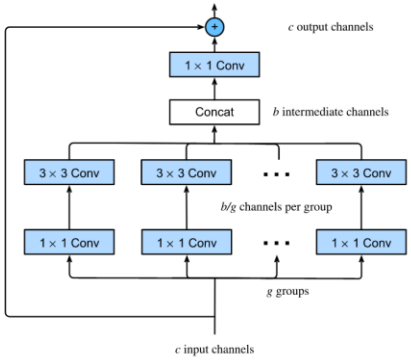

**Residual Neural Networks (ResNet)**, by He et al. in 2016, is a network architecture consisting of blocks of residual blocks. There are different ResNets depending on how many blocks or layer are used. For example, the following is a ResNet-18, as it has 18 trainable layers (the first 7x7 conv, two 3x3 convs for each internal block, and there are eight of them, and the last fully connected layer):



ResNet is a widely used backbone until today.

**ResNexts: Group convolution**, by Xie et al. in 2017. To add more non-linearities in ResNets we can add more layers, increase the width of convolutions or the number of channels. However, increasing the number of channels add quadratic complexity ($F_{in} \times F_{out}$). Group convolutions is a technique to reduce the computational complexity while still increasing the network's width. In group convolution, the input channels are divided into groups, and a convolution is performed independently on each group. This means that if you have $g$ groups, each group will have $\frac{c}{g}$ input channels and $\frac{b}{g}$ filters applied to it, where $c$ is the total number of input channels, and $b$ is the total number of intermediate channels. Each filter in a group only interacts with the input channels within that group.

The following diagram shows a block of the ResNeXt architecture. The input channels are first split into $g$ groups. Each group is processed by its own set of 1x1 convolutions (to reduce dimensionality), followed by 3x3 convolutions (to capture spatial hierarchies). The outputs of these groups are then concatenated, and a final 1x1 convolution is applied to the concatenated result to produce the final output channels.




**DenseNet**, by Huang et al. in 2017. This architecture is based on ResNet blocks, but instead of adding the input and the result of the convolution, it concatenates both of them plus the results of each dense blocks on the channel dimension. It makes use of **transition layers**, which are $1 \times 1$ convolutions after concatenation, to reduce feature dimension.

Some other architectures address different problems. MobileNets, by Howard et al. in 2017 was thought to be deployed on mobiles, it separates convolution in two operations; EfficientNet, by Tan et al. in 2019, uses grid search without computational budget to modify the architecture at best width, depth or input resolution...

CNNs are being slowly replaced in some cases by **Visual Transformers**.

### 5.4.1. Object Detection Problem

The goal of the object detection problem is to find a bounding box that contain an object of interest. It implies two objectives:

- Regression to shape the box.
- Classification to identify the object.

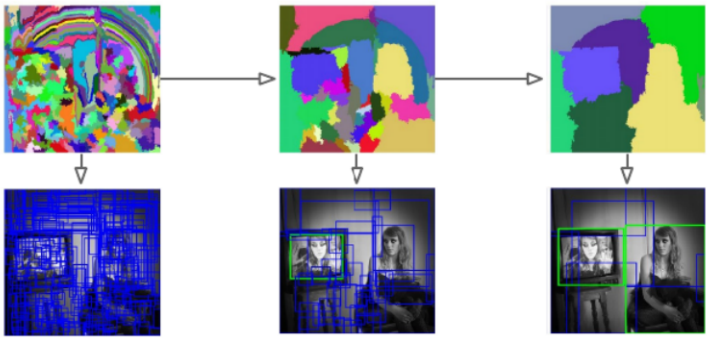
If we consider this problem as a classification problem, one approach is to test various positions and scales to propose various anchor boxes. This is scalable only if the classifier is fast enough. However, CNNs are computationally expensive. A solution is to select only a subset of the boxes, by finding blobs likely to contain objects without class consideration.

**Region proposal: selective search**, by Uijlings et al. in 2013

Selective search is a widely used tool to provide regions of interest in images. The regions are computed following the pipeline:

1. Ascendant segmentation: The algorithm starts with a fine-grained segmentation of the image, where many small regions are created based on pixel similarity (color, texture, etc.). This is often done using a graph-based segmentation method.
2. Fuse regions at different scales: The segmented regions are then hierarchically merged based on various similarity measures. This process is iterative, with smaller regions combining to form larger ones. This step is crucial as it operates over multiple scales, allowing the algorithm to be more robust in detecting objects of different sizes and aspects.
3. Convert regions to potential boxes of interest: The resulting regions from the previous step are then used to generate bounding boxes, which are the "region proposals." These are the candidate regions that could contain objects and are subsequently used by an object detection model for classification.

This process is depicted below:



**Intersection over Union (IoU)**

For two sets, $A$ and $B$, the Jaccard index, or Intersection over Union, is the reatio of the intersection area to their union are

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}.$$

**Label region proposal**

Region proposal anchor boxes are associated to ground-truth bounding boxes given their IoU (above a threshold). The class of an anchor is the same as its associated bounding box. If the anchor box is not associated to a bounding box, it is labeled as background.

The offsets, relative to the position of central coordinates between anchor box and boundingbox, label the region.

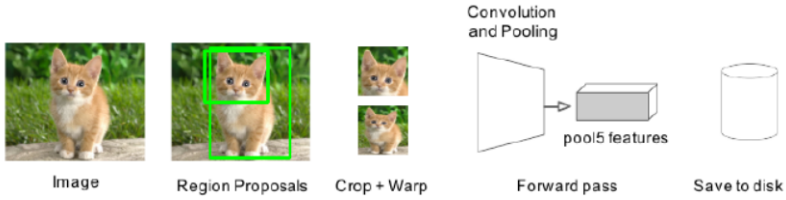
That is, proposal anchor boxes are labeled with the same label of a ground-truth bounding box if their IoU is above a given threshold.

**R-CNN**, by Girshick et al. in 2014

R-CNN takes region proposal as input. It computes features for each of the regions, and the classification is done using several SVMs. The regression of offsets is done by using a linear regression model.

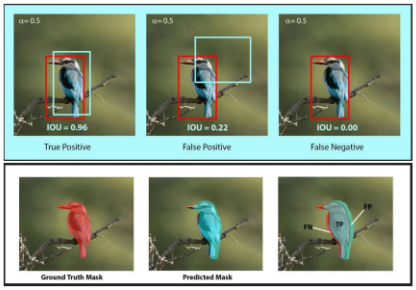
To train R-CNNs, there is a supervised pretraining of a CNN backbone, generally on ImageNet, followed by an optional fine-tuning of the backbone, to learn specialised features based on a classification obkective. The backbone is then fixed.

For each input: crop and wrap proposal regions computed by selective search and compute the features. Train a binary SVM for each class, and apply linear regression to crrect small offsets between the prediction and the actual bounding box.




**Evaluating detection models**

The meaning of true positive, false positive and false negative for the detection problem is not the same as for regular classification problems. The threshold for True and Falso positive/negatives is based on the IoU and a given threshold. For example:




Remember, $precision = \frac{TP}{TP+FP}$ and $recall = \frac{TP}{TP+FN}$.

The **average precision** is the area ander the prevision-recall curve.

The **mean-average precision** is the mean of average precision for various IoU threholds.

R-CNNs is slow in testing phase, because there are as many passes in the CNN as the number of region proposals, which is around 2000. Also, SVM and regression are a bit old school, so this architecture is not usable for real world applications.

**Fast R-CNN**, by Girshick in 2015

This modification of R-CNN was proposed to increase its efficiency. It passes thewhole image only once in the CNN backbone. The CNN is trainable, and not fixed, and the proposed regions are associated with computed features from the output feature map.

Each region of interest can have a different size, so selective search proposed regions are concatenated with features from the CNN to form the **Region of Interest (RoI) Pooling Layer**, which reshape the features to feed them to fully connected layers. From these features the classes and the offsets are predicted.

The following table compares R-CNN and Fast R-CNN:

| | R-CNN | Fast R-CNN |
|---|---|---|
| Test time per image | 47 s | **0.32 s** |
| Speedup | 1x | **146x** |
| Test time per image with selective search | 50 s | **2 s** |
| Speedup | 1x | **25x** |

**Faster R-CNN**, by Ren et al. in 2017

This new modification replaces selective search by a learned region proposal network. The rest is similar to Fast R-CNN.

The **region proposal network (RPN)** is a network learned to propose regions of interest. It slides a window on the feature size, and, at each location of the window, it makes a prediction for $k$ anchors (propositions), which are sampled by varying scale and aspect ratio. A small network predicts if there is an object there, and a small network predicts offsets with the bounding box.

The following table compares the three approaches:

| | R-CNN | Fast R-CNN | Faster R-CNN |
|---|---|---|---|
| Test time per image (with proposals) | 50 s | 2 s | **0.2 s** |
| Speedup | 1x | 25x | **250x** |
| mAP (VOC 2007) | 66.0 | **66.9** | **66.9** |

**You Only Look Once (YOLO)**, by Redmon et al. in 2016

This model is a single stage detector: one network is used to predict the bounding boxes and the class probabilities. It is the first real time detector close to Faster R-CNN performance.

Its architecture consists of 24 convolutional layers and input dimension of $448 \times 448$. For each input, it cuts it in $7 \times 7$ grids of cells. Each cell computes the objectness and anchor boxes, considering the object at the center of the cell, and enabling the boxes to reach out the cell.

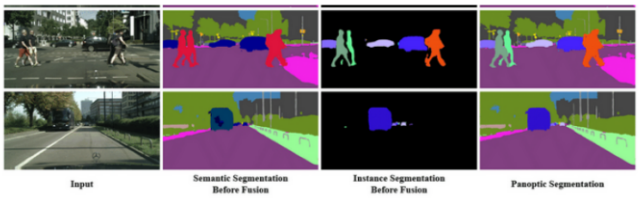
This model has been enhanced multiple times, with YoLo v2, v3, v4, v5,...

### 5.4.2. Semantic Segmentation

The goal in this case is to find an object class for each pixel. There are several associated problems:

- **Semantic segmentation**: associate each pixel to a specific class.
- **Instance segmentation**: associate each pixel to a specific class and identify various instances from the same class.
- **Panoptic segmentation**: associate each pixel to once class, and prevent overlapping segments.

These are illustrated below:




**Mask R-CNN** by He et al. in 2017

Basically, it is Faster R-CNN with a third branch, which outputs the object mask.

It can use various backbones such as ResNet, DenseNet,...

## 5.5. Data and Transfer

Modern CNNs are made of millions of parameters, requiring a huge amount of data to train them. For example, ImageNet has 1.2M images for 1k classes, and is being replaced by ImageNet21k, that has 1B images.

Manually annotating such amount of images is costly in terms of money and time, and therefore there is the need for techniques to deal with lacking of data: data augmentation and transfer learning.

### 5.5.1. Data Augmentation

We already saw it! It increases the diversity of data by transforming the source data with invariants

### 5.5.2. Transfer Learning

Sometimes there is possibility to learn from scratch, by random initialisation, because of lack of data. In this case, we can make use of an already **pre-trained** backbone from a source task for the target task.

- If the target task is similar to the source task, and the size is comparable, transfer learning works directly; while if the size is inferior, there is a risk of overfitting to the target task after few epochs.

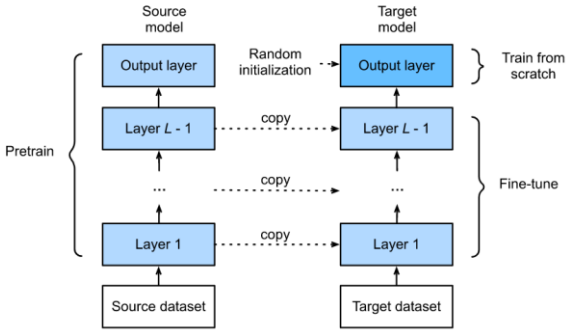
  For small problems, the transfer learning is accomplished by learning a linear classifier with last layers of a pre-trained backbone.

- If the target task is very different from the the source task, and the problem is small, a linear classifier from lower level layers can be used; if the problem is large, we can fine-tune the backbone to the new task.

In all cases, starting from initialized weights is better than nothing.

**Fine tuning**

Fine tuning consists in training several layers of a pre-trained backbone:




**Domain Adaptation**

Sometimes, the target task and the source task share the same classes, but the target data has a different distribution than the source. This is an entire field in ML: how to adapt our features to the new domain, and have the best possible results.

## 5.6. Self-Supervised Learning (SSL)

We already know supervised learning. It consists on predicting a target associated to an input.

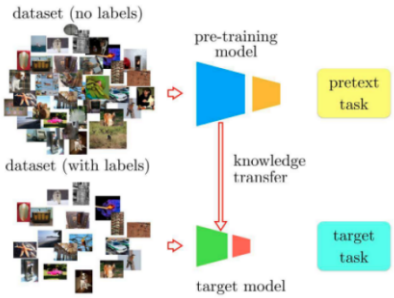
Self-Supervised Learning (SSL) is different. It consists on predicting a part of en input, from the input. This means the model learns to understand and generate data by teaching itself.

A **pretext task** is a task designed to teach the model something about the structure of the input data without using labels. For example, the pretext task might involve predicting the next word in a sentence, or the color version of a grayscale image. The idea is to construct targets from the data itself and learn from these artificially created labels.

SSL presents several advantages:

- Reduces the Need for Labeled Data: Labeling is often expensive and time-consuming, and SSL offers a way to learn useful representations without the need for extensive labeled datasets.
- Better Pre-training: SSL can serve as pre-training for neural networks, allowing them to learn general features from the data that can then be refined with a smaller amount of labeled data for a specific task (using transfer learning).

All this is illustrated below:




For images, several pretext tasks have been firstly proposed, like rotation, patch localization, colorization, counting,...

For videos, which add the time dimension, we find the same pretext tasks as in images, plus some others specific to videos, which can be masking frames, shuffling frames,...

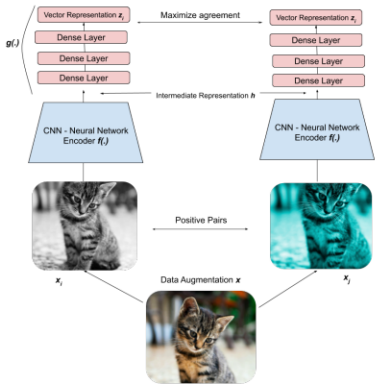
### Contrastive Learning

Constrastive Learning is a pretext task for SSL. It aligns positively pairs of images, and push away other images. We find the problem of defining what is a positive pair, which can be solved by using data augmentation.

The following figure illustrates this concept:

**SimCLR**, by Chen et al. in 2020

This model defines a siamese pipeline for contrastive learning. Positive pairs are formed from strong data augmentation, like color jittering, gaussian blur, grayscale, etc. Both paris go through an encoder and a projector, which consists on several stacked MLP layers. Then, contrastive loss is applied.




**SCE**, by Denize et al. in 2021

This model tries to address the problem of how to deal with negatives that share semantic information with the positive pairs. For example, two different cats share more information than two different dogs. Pushing negatives with high semantic similarities makes training unstable, and so SCE predicts positive pairs and estimated relations among instances.

**Deep Cluster**, by Caron et al. in 2018

Clustering associates several input data to a same pseudo-label, without supervision.

Deep Cluster applies clustering to traing a backbone using several stages:

- Estimate pseudo-labels for each instance with a fixed backbone.
- Train the backbone on this label.
- Repeat the operations until convergence.

# 6. Recurrent Neural Networks

## 6.1. Introduction

Recurrent Neural Networks are a family of Neural Network architectures specially designed to deal with sequential data, such as text in NLP, speech signals in speech2text tasks, temporal data in videos, or other temporal signals, like ECGs, time series, etc.

**Sequential data** is data with a temporal component, that usually implies correlation in the temporal dimension. More precisely, a **sequence** consists in a set of vectors $x_t$, where $t \in [1, \tau]$ is a temporal index.

Note that the temporal index is not necessarily related to time, but to order. For example, text is not temporal, but words are ordered.

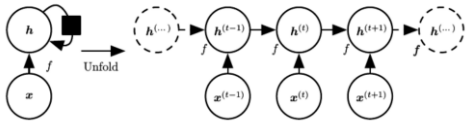
**Memory** is essential for us to understand and interpret what we perceive. Memory can be understood as a persistent format of information. MLP don't have this persistence, each data point is processed independently of the rest. Therefore, RNNs introduce a way to store information, by adding inner loops that enables them to preserve information at each time-step.

## 6.2. Recurrent Neural Networks

A Recurrent Neural Network (RNN) is defined with a state, $h^{(t)}$, by recurrence. This state depends on the current input, $x^{(t)}$, and the previous state, $h^{(t-1)}$:

$$h^{(t)} = f\left(h^{(t-1)}, x^{(t)}; \theta\right).$$

The computational graph can be represented with a loop, or unfolded, making it direct and acyclic, as:




A RNN is trained to predict the future, given past information, in such a way that $h^{(t)}$ is a summary of the sequence until the timestep $t$. This summarization implies information compression, or loss of information, and the training must keep relevant information for the task.

Going back to the definition of the state, we can then write it alternative as

$$h^{(t)} = f\left(h^{(t-1)}, x^{(t)}; \theta\right) = g^{(t)}\left(x^{(t)}, x^{(t-1)}, ..., x^{(1)}\right),$$

where $g^{(t)}$ takes the whole sequence as input, of variable length.

On the other hand, using the factorized form, with **transition function** $f$, the function is the same at each timestep, allowing for parameter sharing and generalization.

**Theorem 6.1.** *RNNs are universal approximators.*

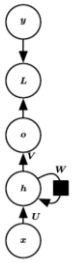
We measure the performance of a RNN through the cost function $L$, between the output $o$ and the ground truth $y$. Internatlly, we use softmax:

$$\hat{y}_i = \frac{e^{o_i}}{\sum_j e^{o_j}}.$$

The parameters of a RNN are:

- $U$: input to hidden layers.
- $W$: recurrent connections between hidden layers.
- $V$: hidden layers to output.

That is:



### 6.2.1. Direct Propagation

In the case of discrete variable prediction (like words), $o$ represents log-likelihoods of possible output values, and the normalize probability is $\hat{y} = softmax\,(o)$.

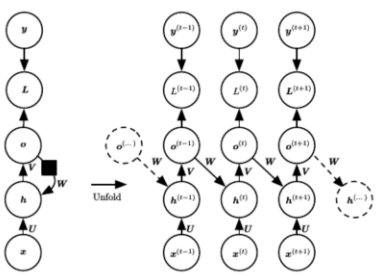
Direct propagation works by:

$$\begin{aligned} a^{(t)} &= b + Wh^{(t-1)} + Ux^{(t)}, \\ h^{(t)} &= \phi\left(a^{(t)}\right), \\ o^{(t)} &= c + Vh^{(t)}, \\ \hat{y}^{(t)} &= softmax\left(o^{(t)}\right), \end{aligned}$$

where $\phi$ is the activation function.

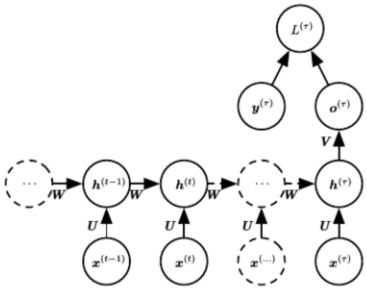
### 6.2.2. Recurrent Neural Networks with output recurrence

We don't link directly hidden layers recurrently, but rather we link the output to a hidden layer. This has the cons of using less information than $h^{(t-1)}$, being therefore a less expressive model, but the pros of being easier to train, enabling parallelization and efficient backpropagation.



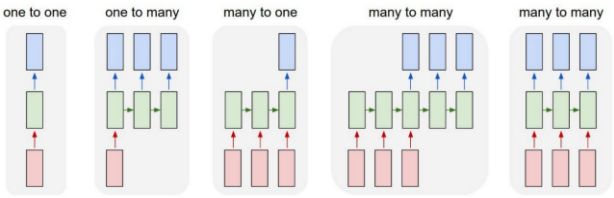

### 6.2.3. Recurrent Neural Networks with unique output

In this case, it needs the complete input sequence, and the output is a summary of the input. This kind of setting is usually used a simpler module of a more complex architecture.



### 6.2.4. More architectures

In this figure we observe different possible setups, which apply to different use cases.

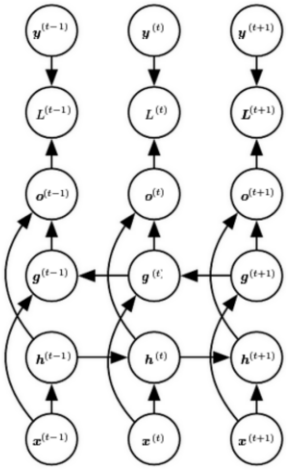



For instance:

- Automatic captioning, converting images to text sequences, could be done with a one to many architecture.
- Sentiment analysis, converting a text sequence to a predicted label, could be done with a many to one architecture.
- Machine translation, converting text sequence to text sequence, could be done with a many to many architecture.
- Frame-label video classification, converting a video sequence into a sequence of labels, could also be done with a many to many architecture.

### 6.2.5. Bi-Directional Recurrent Neural Network

Sometimes, the future of the sequence can also be helpful, for example in NLP, in Character recognition and even in Speech recognition. For this, there exist the Bi-Directional RNN (BRNN), which combine a RNN processing from past to future, with state $h$, and a RNN processing from future to past, with state $g$, and the output combines the two of them.



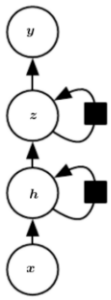
Moreover, BRNN can be applied to images with 4 direction, applying them up,down and left,right. This is a bit outdated approach for visual recognition, with architectures as ReNet for classification and ReSeg for segmentation.

### 6.2.6. Deep Recurrent Neural Network

A classical RNN has an input layer, $U$, a hidden layer, $W$, and an output layer $V$. However, we can add hidden layers, allowing to go higher in abstraction.

The representation improves hierarchically:

- $h$, the first state, represents temporal dependencies on inputs $x$.
- $z$, the second state, represents temporal dependencies on the representations $h$.




Also, we can add an MLP before every recurrent layer, increasing the representation capacity of the network, at the expense of a harder training. A possibility to improve these is to use skip connections.

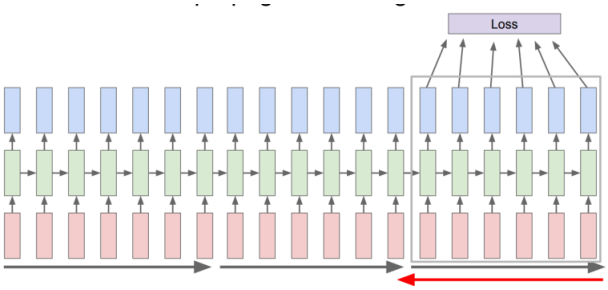
## 6.3. Recurrent Neural Networks Training

The principle is the same as with MLP and CNNs: we choose a cost function a minimize it during training using gradient descent with backpropagation.

However, we need to take into account the recurrency of the network, and so we use what is called **Backpropagation through time** (BPTT), which consists in applying backpropagation to an unrolled graph of the RNN.

The idea is to forward through the entire sequence to compute the loss, and then go backwards through the entire sequence to compute gradient.

However, this can be very inefficient, so we also find the **truncated BPTT**, which does BPTT to chunks of the sequence, instead of the complete sequence. The hidden states are carried forward in time until the end of the sequence, but we only backpropagation for some smaller number of steps.




## 6.4. Long-Short Term Memory (LSTM) and Gated Recurrent Units (GRU)

Classical RNN present two main limitations:

1. A computation limitation: they are hard to train, and prone to gradient vanishing and explosion.
2. Long term interactions are hard to model.

Some upgraded variants of RNNs are Long Short Term Memory (LSTM) and Gated Recurrent Units (GRU).

**Gradient vanishing and exploding**: the reason RNNs are prone to gradient vanishing/exploding is that the gradient needs to go through the unfolded computational graph. If the largest singular value of the hidden matrix is greater than 1, a recurrent multiplication by it would tend to increase the gradient vector, leading to gradient explosion. The opposite happens when the largest singular value is smaller than 1, leading to gradient vanishing.

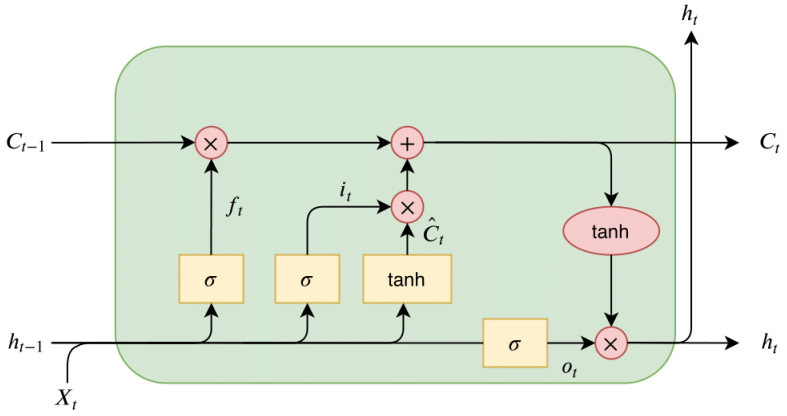
The solution to gradient explosion can be using gradient clipping.

The solution to gradient vanishing is to change the RNN architecture.

**Long-term dependency problem**: RNNs can connect past information to present data, and, in theory, time distance of information is irrelevant. However, in practice, classical RNNs cannot model long sequences (more than 20 points).

### 6.4.1. LSTM

LSTM is based on a standard RNN whose neuron activates with *tanh*.




$C_t$ is the cell state, which flows through the entire chain and is updated with a sum instead of a product. This avoids memory vanishing and the gradient explosion and vanish.

Then, there are three gates foverned by sigmoid units, which define the control of in and out information.

A key component is the forget gate,

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right),$$

which can affect the affect the previous state, letting it through or not (forgetting).

Then, the input gate layer,

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right),$$

which processes the input, contributing to the cell state.

Then, there is the classical neuron, activated with *tanh*, which also contributes to the cell state, by

$$\hat{C}_t = tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right).$$

Finally, to update the cell state it is

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \hat{C}_t.$$

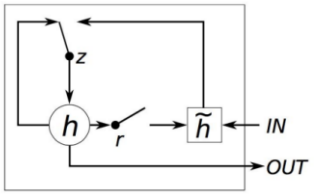
Notice that the output is computed as

$$h_t = tanh\left(C_t\right) + \sigma\left(W_o \cdot [h_{t-1}, x_t] + b_o\right).$$

### 6.4.2. GRU

GRU is a variant of LSTM, simpler and faster, and with similar performance. It works by applying:

$$\begin{aligned}
u_i &= \sigma\left(W_u \cdot x_i + U_u \cdot h_{i-1} + b_u\right), \\
r_i &= \sigma\left(W_r \cdot x_i + U_r \cdot h_{i-1} + b_r\right), \\
\hat{h}_i &= tanh\left(W_h \cdot x_i + r_i \circ U_h \cdot h_{i-1} + b_h\right), \\
h_i &= u_i \circ \hat{h}_i + (1 - u_i) \circ h_{i-1}.
\end{aligned}$$




Even with these improvements, RNN struggle with long sequences, and remain difficult and slow to train, because of thei autoregressive nature. In 2017, a Google paper introduced a novel architecture that changed everything: the Transformer.

## 7. Deep Generative Modelling

Recall that the objective of supervised learning is to learn a mapping, $f$, between a data sample, $x$, and its label, $y$:

$$y = f\left(x\right).$$

This can be formalized by learning the conditional distribution between labels, given a data point, that is, $p\left(y|x\right)$.

For this task, we need to have access the annotated pairs $(x, y)$. The applications of supervised learning are classification, object detection, segmentation, etc.

On the other hand, unsupervised learning tries to learn the underlying hidden data structure, formalized by the unconditional distribution data points, $p\left(x\right)$.

In this case, we don't need any annotations to the data. The applications of unsupervised learning are generative modelling, clustering, feature learning, summarization, etc.

**Generative modelling** consists in learning a model, $M$, that can approximate the data distribution,

$$p_M\left(x\right) \sim p\left(x\right),$$

and should enable sampling easily. That is, a generative model aims to provide:

- Density estimation: it can evaluate the likelihood of a data point belonging to the original distribution.
- Sampling: it can provide likely samples.

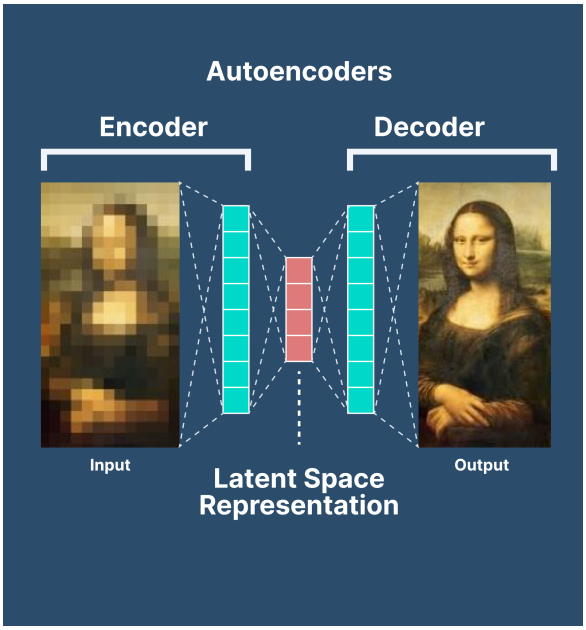



Figure 3: Illustration of autoencoders. Source: https://www.v7labs.com/blog/autoencoders-guide.

## 7.1. Variational Auto-Encoders (VAE)

### 7.1.1. Auto-Encoders

AEs are a type of unsupervised model used to learn low-dimensional latent representations, $z$, of data samples, $x$. This means that the aim is to somehow try to summarize the input data space, $X$, into a smaller space, $Z$, so $dim(Z) < dim(X)$. $Z$ should capture relevant characteristics of data, and the choice of its dimensionality is a compromise between interpretability and the amount of information to keep.

In Figure Figure 3 we can see an illustration of autoencoders. The input is processed by the encoder, which is able to embed it into a smaller space, the **latent space**. The decoder can take the point in the latent space and reconstruct the original image.

To train them, this process is applied to the training set, and the loss function is the distance between the reconstructed images and the original ones:

$$\mathcal{L}(M, X) = \sum_{x \in X} \|M(x) - x\|^2 .$$

The problem of AEs is that they don't allow generation, because the distribution of $p(z)$ is difficult to sample from.

To see this more clearly, let's discuss an example done by Joseph Rocca, Understanding VAEs.

In the following image, we observe how, when using AEs, points in the latent space can have no real meaning, making it difficult to sample and generate new objects, which is the ultimate goal of generative models.

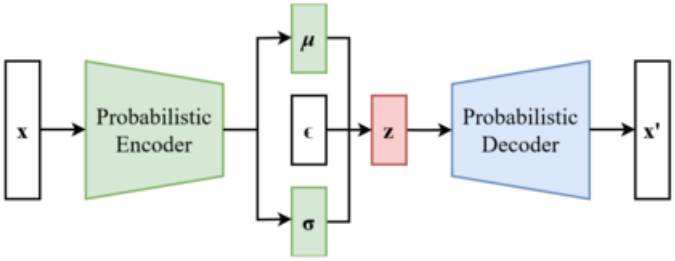
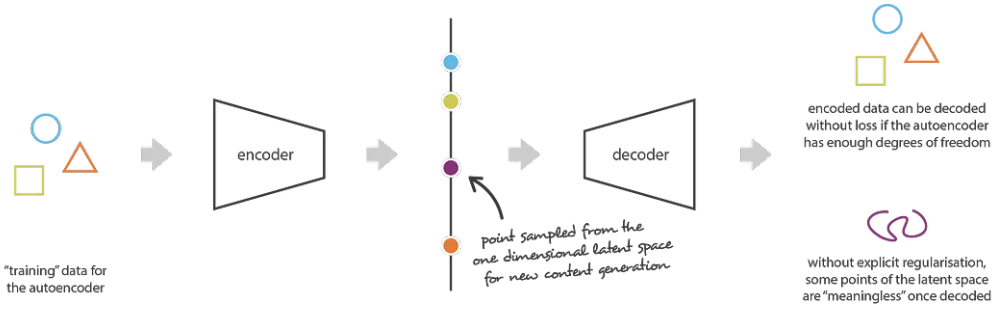



Figure 4: Illustration of a variational auto-encoder.. Source: Wikipedia.




What is desirable, is that somehow, points in the latent space that are close to each other, should have a similar real meaning, once they are decoded:


This cannot be achieved with AEs, and this led to the introduction of Variational Auto-Encoders (VAEs).

### 7.1.2. Variational Auto-Encodersi

In this case, the latent representation is not directly learnt, but rather a **latent distribution**. In Figure Figure 4, we observe the illustration of VAEs.

In this case, training is more complex. We want to mazimize the data likelihood. More precisely, its log-likelihood:

$$\mathcal{L}(\theta, \phi, x) = \log p_\theta(x).$$

Notice that this is not dependent on the distribution of $z$, $q_\phi$, so we can write:

$$\log p_\theta(x) = \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log p_\theta(x)\right],$$

and applying the reversed Bayes' Rule:

$$\mathbb{E}_z\left[\log p_\theta(x)\right] = \mathbb{E}_z\left[log\frac{p_\theta(x|z)\,p_\theta(x)}{p_\theta(z|x)}\right].$$

Here, we can multiply and divide by $q_\phi(z|x)$ and use the properties of logarithms, to get:

$$\mathbb{E}_z\left[\log p_\theta(x)\right] = \mathbb{E}_z\left[\log p_\theta(x|z)\right] - \mathbb{E}_z\left[\log\frac{q_\phi(z|x)}{p_\theta(z)}\right] + \mathbb{E}_z\left[\log\frac{q_\phi(z|x)}{p_\theta(z|x)}\right].$$

Now, recalling the definition of the Kullback-Leibler Divergence:

$$D_{KL}(P\parallel Q) = \mathbb{E}_P\left[\log\frac{P(x)}{Q(x)}\right],$$

we can rewrite:

$$\mathcal{L}(\theta,\phi,x) = \mathbb{E}_z\left[\log p_\theta(x|z)\right] - D_{KL}\left(q_\phi(z|x)\parallel p_\theta(z)\right) + D_{KL}\left(q_\phi(z|x)\parallel p_\theta(z|x)\right).$$

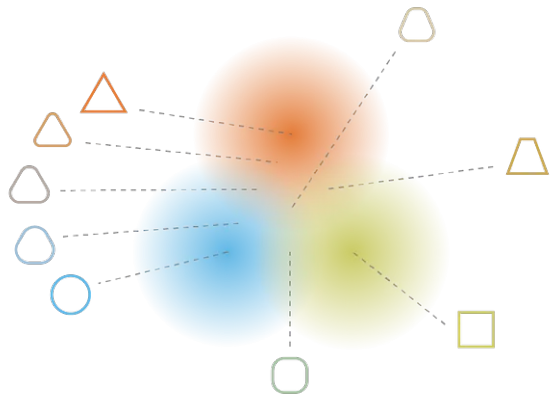
Now, the last term is very difficult to compute, because $p_\theta(z|x)$ is intractable. Since it is bigger than 0, we can bound $\mathcal{L}$ from below by

$$\mathcal{L}(\theta,\phi,x) \geq \mathbb{E}_z\left[\log p_\theta(x|z)\right] - D_{KL}\left(q_\phi(z|x)\parallel p_\theta(z)\right),$$

this is called the **Evidence Lower Bound**. The first term is called the **reconstruction term**, and can be approximated by sampling and serves to ensure good reconstructions of the output. The second term is the **regularisation term**, which makes the distribution of the encoder close to the prior.

The impact of the regularisation is that it enables for **continuity** in the latent representation, so that similar data samples have similar latent representations; and it helps to ensure **completeness** of the latent space, so that all latent points have a meaningful decoding.

Joseph Rocca illustrates this as:



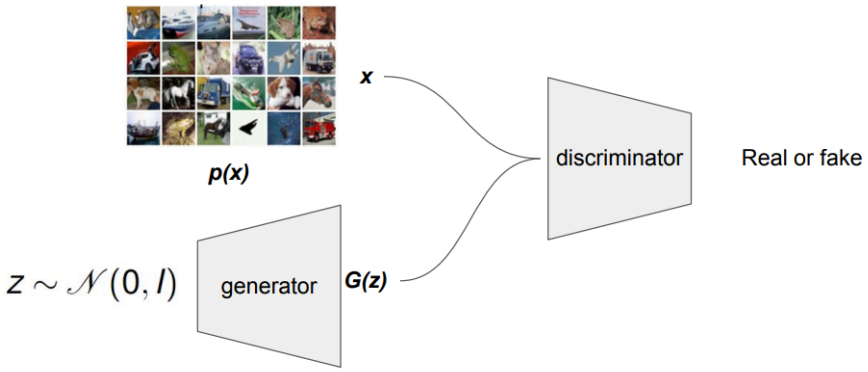
In addition to enabling continuity and completeness, or maybe thanks to this, VAEs enable a much easier sampling and generation. We can just sample $z$ from a normal distribution, and process it through the decoder.

## 7.2. Generative Adversarial Networks

Now, the objective is to sample for a high dimensional and multi-modal distribution, $p(x)$, by learning a mapping, $G$, between a simple distribution from which we can sample and the high dimensional distribution. For this, Generative Adversarial Networks (GANs), consist of two components:

- **Discriminator**: tries to distinguish between generated and real data.
- **Generator**: tries to fool the discriminator by creating likely samples.

This is illustrated below:




To train these models, we follow a minimax approach: the discriminator aims at maximizing its success rate, while the generator aims to minimizing the success rate of the discriminator. This can be expressed as:

$$\begin{aligned}\mathcal{J}(D,G,X) &= \min_G \max_D V(D,G,X) \\ &= \min_G \max_D \mathbb{E}_{x\sim p_{data}(x)}\left[\log D(x)\right] + \mathbb{E}_{z\sim p_z(z)}\left[\log\left(1-D\left(G\left(z\right)\right)\right)\right].\end{aligned}$$

This can be divided:

- The **discriminator loss** is

$$\mathcal{J}(D,X) = \max_D \mathbb{E}_{x\sim p_{data}(x)}\left[\log D(x)\right] + \mathbb{E}_{z\sim p_z(z)}\left[\log\left(1-D\left(G\left(z\right)\right)\right)\right].$$

- The **generator loss** is

$$\mathcal{J}(G,X) = \min_G \mathbb{E}_{z\sim p_z(z)}\left[\log\left(1-D\left(G\left(z\right)\right)\right)\right].$$

Once the training is done, generation is quite straightforward. We only need to sample $z \sim p_z(z)$ and process it with the generator module.

An interesting property of GANs is that they show continuous representations of the latent space, as shown in the figure below:

In the following table, we compare GANs and VAEs in different aspects:

| | Sample Quality | Inference | Mode Coverage | Training Quality Assesment | Training Stability |
|---|---|---|---|---|---|
| VAE | Intermediate | Good | Good | Good | Good |
| GAN | Very good | Bad | Intermediate | Intermediate | Intermediate |

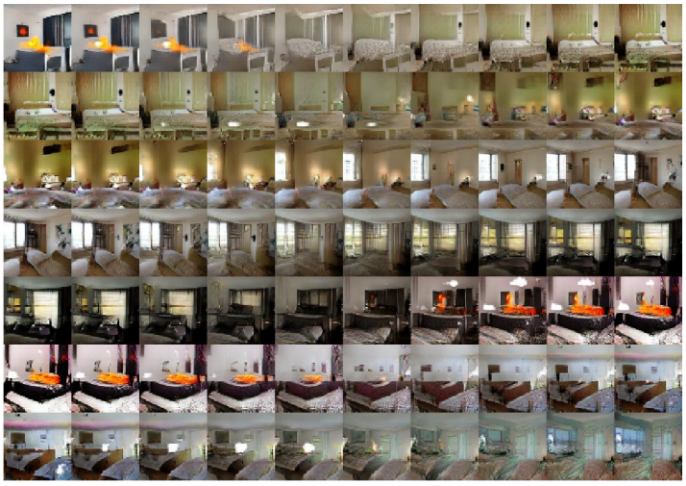


Figure 5: Interpolating in the latent space of GANs. Source: [2].

## 7.3. Other Approaches

### 7.3.1. Normalizing Flows

The idea is to transform a simple distribution into acomplex one, by applying a sequence of invertible transformation functions.

The pros of this approach is that it provides good sample quality, exact inference and stable training. However, it requires invertible neural networks.

### 7.3.2. Pixel RNN

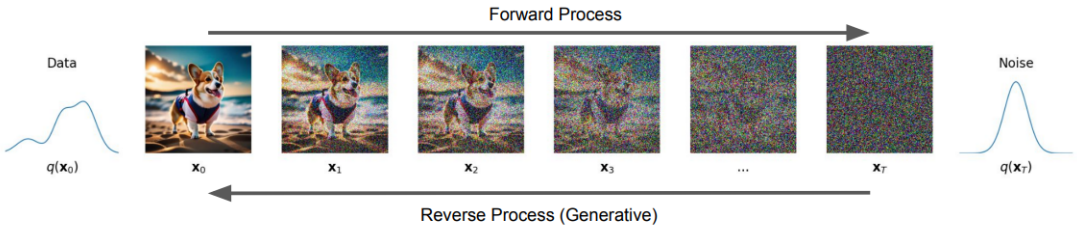
This approach consists in generated image pixels starting from the corner. The dependencies between the current pixel and the previous ones is modelled using an RNN. The biggest drawback of this approach is that sequential generation is very slow.

# 8. Denoising Diffusion Models

To train diffusion models, there are two processes involved:

- Forward process: gradually adds noice to the input, until obtaining a completely noisy image.
- Reverse process: learns to generate data by denoising.

This is illustrated in this Figure:



The forward process can be seen as a Markov process, with starting state $x_0$ and transition probability $q(x_t|x_{t-1})$ for $t = 1, ..., T$:

$$x_0 \overset{q(x_1|x_0)}{\rightarrow} x_1 \overset{q(x_2|x_1)}{\rightarrow} x_2 \overset{q(x_3|x_2)}{\rightarrow} ... \overset{q(x_{T-1}|x_{T-2})}{\rightarrow} x_{T-1} \overset{q(x_T|x_{T-1})}{\rightarrow} x_T$$

The transition probability or **Markov kernel** is usually chosen to be

$$q(x_t|x_{t-1}) = \mathcal{N}\left(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t I\right), \ \beta_t \in (0,1),$$

but it can be any Markov Diffusion Kernel. The property of these kernels is that they converge to a standard distribution. In this case, they converge to a normal distribution, always the same, no matter the input, $x_0$.

Thus, the forward trajectory can be expressed as

$$q(x_{0;T}) = q(x_0) \prod_{t=1}^{T} q(x_t|x_{t-1}).$$

And to go to step $t$, from the step $t-1$, we just need to sample from the kernel distribution.

It is also possible to go advance several steps at once. For this, we can define the parameters:

$$\begin{cases} \alpha_t = 1 - \beta_t \\ \hat{\alpha}_t = \prod_{s=1}^{t} \alpha_s \end{cases},$$

and use the Markov kernel

$$q(x_t|x_0) = \mathcal{N}\left(x_t; \sqrt{\hat{\alpha}_t}x_0, (1-\hat{\alpha}_t) I\right),$$

which is equivalent to do

$$x_t = \sqrt{\hat{\alpha}_t}x_0 + \sqrt{1-\hat{\alpha}_t}\varepsilon$$

with $\varepsilon \sim \mathcal{N}(0, I)$. This allows to go to any node in the chain in just one step.

The parameter $\beta_t$ is the **noise scheduler**, and it's chosen to have increasing variance, in such a way that $q(x_T)$ is a standard Gaussian. If the $\beta_t$-s are small enough, the reverse transitions are also Gaussian, but $q(x_{t-1}|x_t)$ in intractable, but using the Bayes rule we know that

$$q(x_{t-1}|x_t) \propto q(x_{t-1}) q(x_t|x_{t-1}),$$

and we can use a model to approximate this reverse process.

That is, we are going to train a model to approximate the reverse, denoising, process. This parametric reverse model:

- Is also a Markov chain, starting at $x_T$.
- It starts with $p(x_T) = \mathcal{N}(x_T; 0, I)$, since $q(x_T) \approx \mathcal{N}(0, I)$.
- Its transitions are learned.

So, how may we achieve this?

The reverse process is:

$$x_0 \overset{p_\theta(x_0|x_1)}{\leftarrow} x_1 \overset{p_\theta(x_1|x_2)}{\leftarrow} x_2 \overset{p_\theta(x_2|x_3)}{\leftarrow} ... \overset{p_\theta(x_{T-2}|x_{T-1})}{\leftarrow} x_{T-1} \overset{p_\theta(x_{T-1}|x_T)}{\leftarrow} x_T$$

So the reverse transitions are

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}\left(x_{t-1}; \mu_\theta(x_t, t), \sigma_t^2 I\right),$$

and $\mu_\theta(x_t, t)$ will be our trainable network! The reverse trajectory is then

$$p_\theta(x_{0:T}) = p(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1}|x_t).$$

To train this, we follow a maximum likelihood approach:

$$\mathcal{J}(x, \theta) = -\log p_\theta(x_0)$$

Now, we can marginalize by integrating over the latents:

$$-\log p_\theta(x_0) = -\log \int p_\theta(x_{0:T})\, dx_{1:T},$$

which is a shortened notation for $-\log \int ... \int p_\theta(x_0, x_1, ..., x_T)\, dx_1 ... dx_T$. Here, we can multiply and divide by $q(x_{1:T}|x_0)$:

$$-\log p_\theta(x_0) = -\log \int p_\theta(x_{0:T}) \frac{q(x_{1:T}|x_0)}{q(x_{1:T}|x_0)} dx_{1:T},$$

which can be seen as an expectation:

$$-\log p_\theta(x_0) = -\log \mathbb{E}_{x_{1:T}\sim q(x_{1:T}|x_0)} \left[ \frac{p_\theta(x_{0:T})}{q(x_{1:T}|x_0)} \right],$$

and in this point we can use the Jensen's inequality:

$$-\log \mathbb{E}_{x_{1:T}\sim q(x_{1:T}|x_0)} \left[ \frac{p_\theta(x_{0:T})}{q(x_{1:T}|x_0)} \right] \leq \mathbb{E}_{x_{1:T}\sim q(x_{1:T}|x_0)} \left[ -\log \frac{p_\theta(x_{0:T})}{q(x_{1:T}|x_0)} \right] = L_{VLB}.$$

We have reached $L_{VLB}$, the **variational lower bound**. After some derivations, it is possible to arrive to

$$\mathbb{E}_q \left[ D_{KL}(q(x_T|x_0) \| p(x_T)) + \sum_{t>1} D_{KL}(q(x_{t-1}|x_t, x_0) \| p_\theta(x_{t-1}|x_t)) - \log p_\theta(x_0|x_1) \right],$$

here $D_{KL}(q(x_T|x_0) \| p(x_T))$ has no learnable parameters, so we can remove it, and we name:

$$L_{t-1} = D_{KL}(q(x_{t-1}|x_t, x_0) \| p_\theta(x_{t-1}|x_t))$$

and

$$L_0 = -\log p_\theta(x_0|x_1).$$

Now, $D_{KL}(P\|Q)$ is always non-negative, and is 0 when $P = Q$. Also, when conditioned on $x_t$ and $x_0$, the reverse process becomes tractable:

$$q(x_{t-1}|x_t, x_0) = \mathcal{N}\left(x_{t-1}; \hat{\mu}_t(x_t, x_0), \hat{\beta}_t I\right),$$

where $\hat{\mu}_t(x_t, x_0) = \frac{\sqrt{\hat{\alpha}_{t-1}}\beta_t}{1-\hat{\alpha}_t} x_0 + \frac{\sqrt{\alpha_t}(1-\hat{\alpha}_{t-1})}{1-\hat{\alpha}_t} x_t$ and $\hat{\beta}_t = \frac{1-\hat{\alpha}_{t-1}}{1-\hat{\alpha}_t}\beta_t$.

Of course, we don't have $x_0$ available at inference time, so what we want is $p_\theta(x_{t-1}|x_t)$ to be as close as possible to $q(x_{t-1}|x_t, x_0)$. For this, we use the normal law and approximate our $\hat{\mu}_t(x_t, x_0)$ in our approximated reverse process.

## 9. Transformers

Credits to Lucas Beyer's slides https://acdl2023.icas.cc/dr-lucas-beyers-slides/.

## 9.1. Attention

Attention is a function, similar to a *soft* k-v dictionary lookup. Let's delve into it more precisely:

1. **Attention weights**, $a_{1:N}$ represents query-key similarities. They are defined by

$$\hat{a}_i = q \cdot k_i,$$

and normalized via softmax

$$a_i = \frac{e^{\hat{a}_i}}{\sum_j e^{\hat{a}_j}}.$$

2. **Output**, $z$, is an attention-weigthed average of the values $v_{1:N}$. That is:

$$z = \sum_i \hat{a}_i v_i = \hat{a} \cdot v.$$

3. Usually, $k$ and $v$ are derived from the same input $x$:

$$k = W_k \cdot x,$$

$$v = W_v \cdot x.$$

The query, $q$, can come from a separate input $y$:

$$q = W_q \cdot y$$

or from the same input $x$ (in which case it's called **self-attention**):

$$q = W_q \cdot x.$$

In addition, it is usual to use many queries, $q_{1:M}$, not just one... Stacking the different queries leads to the definition of the **attention matrix**, $A_{1:N,1:M}$, and subsequently to many outputs:

$$z_{1:M} = Attn\left(q_{1:M}, x\right) = \left[Attn\left(q_1, x\right)|...|Attn\left(q_M, x\right)\right].$$

We usually use **multi-head attention**. This simply means that the operation is repeated $K$ times, with results concatenated along the feature dimension, with different $W_k$s.

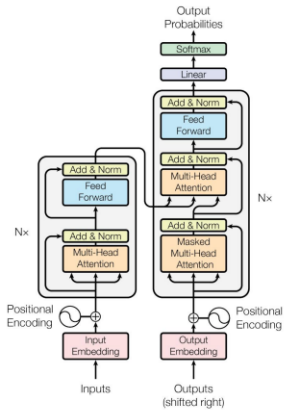
The most commonly seen formulation of the process is

$$z = softmax\left(\frac{QK'}{\sqrt{d_{key}}}\right)V,$$

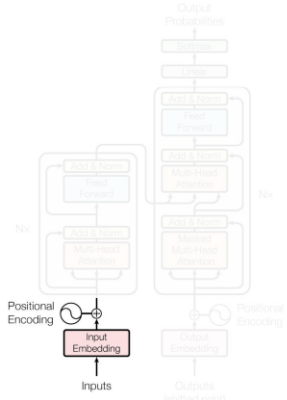
where here actually $Q$ is the query matrix, $K$ is the key matrix, and $V$ is the value vector. $d_{key}$ is the dimension of a key in $K$.

## 9.2. Architecture

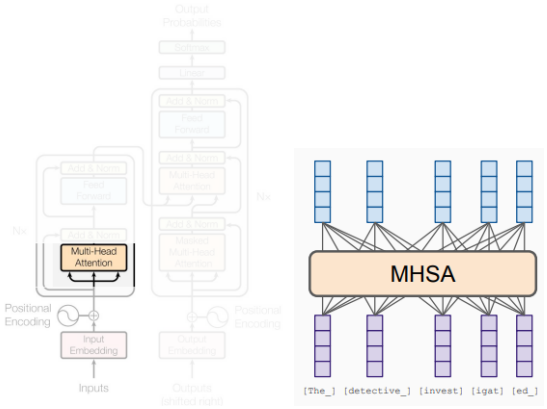
The architecture of the transformer is the following:



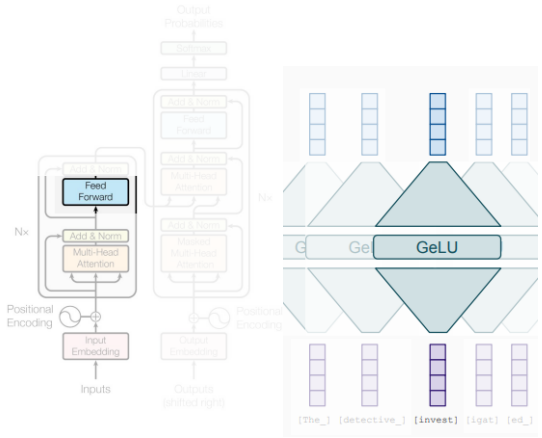

Let's analyze it step by step:



- **Input tokenization and embedding**:
  - The input text is first split into pieces, called tokens. These can be characters, words,... or even bytes.
  - Tokens can be represented as indices of a dictionary of all possible tokens.
  - Each entry corresponds to a learned $d_{model}$-dimensional vector in the embedding space.
- **Positional encoding**:
  - Attention is permutation invariant, but language is not.
  - Therefore, we need to provide a way to differenciate the position of each word. The idea is to add something based on the position of the token in the sentence. For example, add 1 for the first position, 2 for the second, and so on. In reality, a more complex approach is used, but with the same idea in mind.



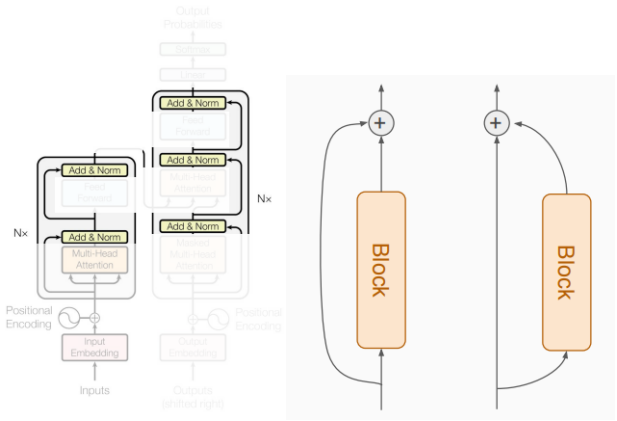

- **Multi-headed self-attention**:
  - The input sequence is used to create queries, keys and values.
  - Each token can *look around* the whole input sequence, and decide how to update its representation based on what it sees. For example, the word *seed* will have a different representation in a biology context and in a computer science context.



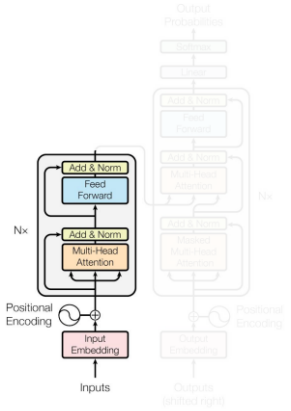

- **Point-wise MLP**
  - A simple MLP is applied to each token individually, with GeLU activation function:

$$z_i = W_2 \cdot GeLU\left(W_1 x + b_1\right) + b_2.$$

  - This is as each token pondering for itself about what it has observed previosly.
  - There is some weak evidence this is where the world knowledge is stored.
  - It contains the bulk of the parameters of the model. When people make giant models and sparse/moe, this is what becomes giant.
  - Some people like to call it a $1 \times 1$ convolution.

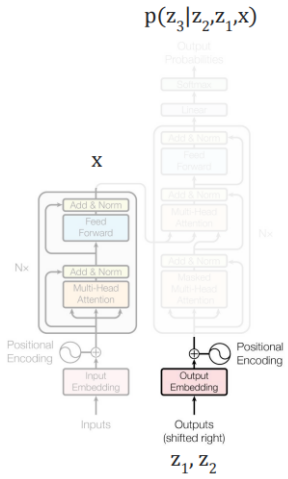



- **Residual connections**:
  - Each module's output has the exact same shape as its input.
  - Following ResNets, the module computes a residual, instead of a new value. That is, the action of module $M$ is
$$z_i = M(x_i) + x_i.$$
  - Was shown to dramatically improve trainability.
- **LayerNorm:**
  - Normalization also improves trainability.
  - There is post-norm (the one used originally)
$$z_i = norm(M(x_i) + x_i),$$
    and pre-norm (a modern approach)
$$z_i = M(norm(x_i)) + x_i.$$




- **Encoding/Encoder**:
  - Since input and output shapes are identical, we can stack $N$ such blocks.
  - Typically, $N = 6$ is base and $N = 12$ is large. But more can be stacked.
  - Encoder output is a *heavily processed* version of the input tokens.

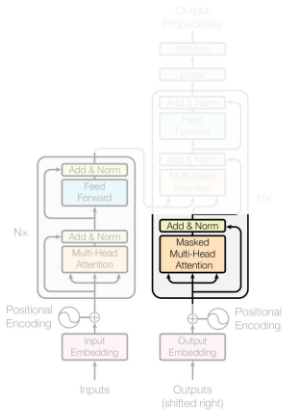
– This has nothing to do with the requested output yet. This is achieved at the decoder.




- **Decoding/Decoder**:
  - We want to model $p(z|x)$, to select the most likely outcome.
  - This seems impossible at first... but we can decompose this into tokens:

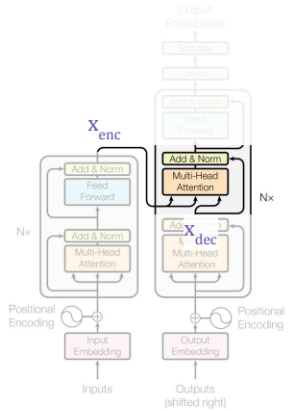
$$p(z|x) = p(z_1|x)\, p(z_2|z_1, x)\, p(z_3|z_2, z_1, x) \cdot ...$$

    Meaning, we can generate the answer one token at a time, where each $p$ is a full pass through the model.
  - For generating $p(z_1|z_2, z_1, x)$:
    1. $x$ comes from the encoder
    2. $z_1, z_2$ is what we have predicted so far, goes into the decoder.
  - Once we have $p(z|x)$, we actually need to actually sample a sentence. There are different strategies for this, like greedy or beam-search, among others.




- **At training time: masked self-attention**:
  - This is regular self-attention, as in the encoder, to process what's been decoded so far ($z_1$ and $z_2$ in the previous example), but with a trick.

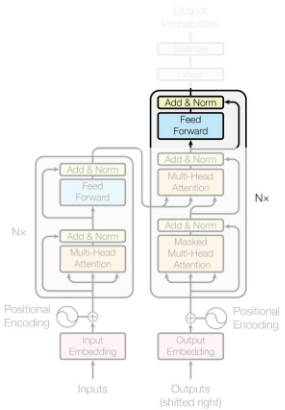
  - If we had to train on one single $p(z_3|z_2, z_1, x)$ at a time, the process would be very slow.
  - Instead, we can train on all $p(z_i|z_{1:i-1}, x)$ simultaneously.
  - For this, in the attention weights for $z_i$, we set all entries $i : N$ to 0, and use the ground truth $y$ to simulate that we have already computed the previous tokens of $z$.
- **At generation time**:
  - This trick is not possible now. We need to generate each $z_i$ at a time.
  - This is why autoregressive encoding is extremely slow.



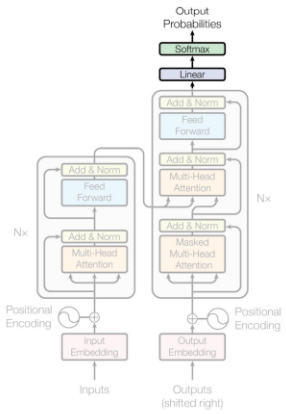

- **Cross-attention**:
  - Each decoded token can look at the encoder's output:

$$Attn\left(q = W_q x_{dec}, k = W_k x_{enc}, v = W_v x_{enc}\right).$$

  - This is where the conditional $|x$ in $p(z_i|z_{1:i-1}, x)$ comes from.
  - Since self-attention is so widely used, people have started to call it attention, and cross attention, which is in fact the original attention, is usually called cross-attention.

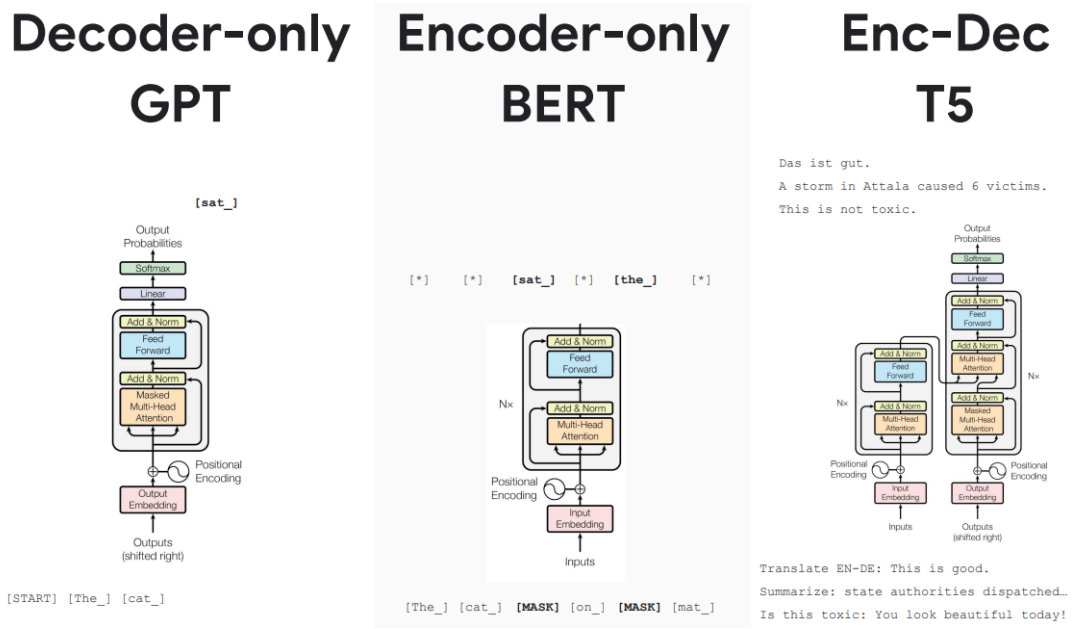



- **Feedforward and stack layers**: similar as before.




- **Output layer**:
  - Assume we have already generated $K$ tokens, we generate the next one.
  - The decodeer was used to gather all information necessary to predict a probability distribution for the enxt token over the whole vocabulary.
  - It is fairly simple: a linear projection of token $K$ and a $softmax$ normalization to select the output.

## 9.3. The first big takeover: Language Modeling




## 9.4. The second big takeover: Computer Vision

Previous approaches used attention by:

- Applying it at the pixel level, but locally or factorized. The results are usually not amazing, definitely not justifying the increased complexity nor the slowdown over convolution networks.

- Globally, after or inside a full-blown CNN, or even in a detector/segmenter architecture. The results were highly complex, often multi-stage trained architecture. Also, the transformer can't learn to fix the CNN's mistakes.

Then came the **Vision Transformer**. Many prior works attempted to introduce self-attention at the pixel level. For $224px^2$, this is a 50k sequence length. Definitely too much. Therefore, most works restrict attention to local pixel neighbourhoods, or as high-level mechanism on top of detections.

The key breakthrough for using the full transformer architecture, standalone, was to tokenize the images by cutting it into patches of 16x16 pixels, and treating each patch as a token... i.e., embedding it into an input space.

## 9.5. The third big takeover: speech

The **conformer** is a modified architecture, following a same story as happened in the vision realm, but with spectograms of the sounds instead of images.

The conformer adds a third type of block, using convolutions, and slightly reorder blocks, but overal is very similar to transformer.

It exists as encoder-decoder, or as encoder-only variant with CTC loss.

## 9.6. The fourth big takeover: Reinforcement Learning

The **decision transformer** casts the supervised/online RL problem into a sequence modeling task. It can generate/decode sentences of actions with desired return. The trick is prompting:

*"The following is a trajectory of an expert player: [obs] ..."*

## 9.7. The Transformer's unification of communities

Anything we can tokenize, we can feed to Transformer! The idea is basically to tokenize different modalities, each one in their own way, and send them all jointly into a Transformer... And this seems to work.

This is called **multimodal model**, and there is currently an explosion of works doing this.

## 9.8. A note on efficient transformers

The self-attention operation complexity is $O\left(N^2\right)$ for sequence length $N$, because we need to compute a matrix product of size $N$. We'd like to use larger $N$, for whole articles or books, full video movies, high resolution images, etc.

Many $O\left(N\right)$ approximations to the full self-attention have been proposed, but none provides yet a clear improvment. There seems to always be a trade-off between speed and quality.


Big Bird
Transformer
Synthesizer
Linformer
Performer
Reformer
Sinkhorn
Linear Transformer
Local Attention
LRA Score
Speed (examples per sec)
56
54
52
50
48
46
44
50
100
150
200
250
300
350

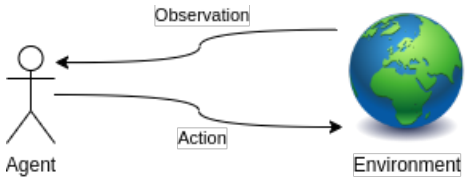
# Part II.
# Reinforcement Learning

## 10. 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 10.1.** *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 10.1.** 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.

# 11. Definition and components

**Definition 11.1.** 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!

## 11.1. 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:

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

## 11.2. Markov Decision Processes

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

**Definition 11.2.** 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 11.1.** 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: $S_t = u\left(S_{t-1}, A_{t-1}, R_t, O_t\right)$ (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.

## 11.3. 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,

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

or stochastic,

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

## 11.4. 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

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

The value depends on the policy, $\pi$, 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

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

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

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

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

### 11.4.1. 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.

## 11.5. 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:

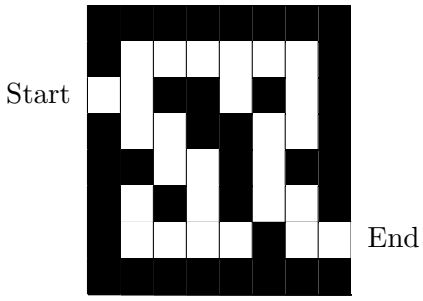
$$\mathcal{P}(s, a, s') \approx p(S_{t+1} = s' | S_t = s, A_t = a).$$

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

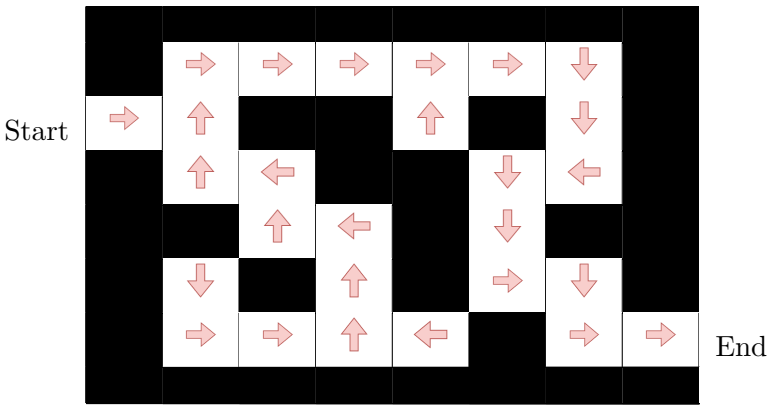
$$\mathcal{R}(s, a) \approx \mathbb{E}[R_{t+1} | S_t = s, A_t = a].$$

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.

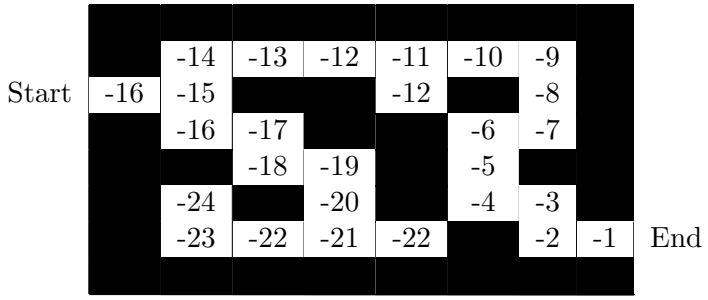
**Example 11.2.** Consider the following maze, where the rewards are -1 per time-step, the actions are to go N, E, S and W, and the states are the agent's location:




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.

## 11.6. 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.

## 11.7. 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.

# 12. Markov Decision Processes

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

**Definition 12.1.** A **Markov Decision Process (MDP)** is a tuple $(S, A, p, \gamma)$ 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* 12.1. $p$ 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 12.2.** A MDP is a tuple $(S, A, p, r, \gamma)$ 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 12.3.** 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 12.1.** In an MDP, which of the following statemest are true?

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

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

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

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

and there is the following connection between them:

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

Also, we can define the maximum possible value functions:

**Definition 12.4.** The **optimal state value function**, $v^*\left(s\right)$, is the maximum value function over all policies,

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

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

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

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

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

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

**Theorem 12.1.** ***Optimal Policies Theorem***
*For any MDP:*

- *There exists an optimal policy $\pi^*$ 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* 12.2. 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 12.2.** ***Bellman Expectation Equations***
*Given an MDP, $M = (S, A, p, r, \gamma)$, for any policy $\pi$, 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 12.3.** ***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* 12.3. 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.

## 12.1. Solving Reinforcement Learning Problems with Bellman Equations

Pb1. Estimating $v_\pi$ or $q_\pi$ is called **policy evaluation**, or **prediction**:

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

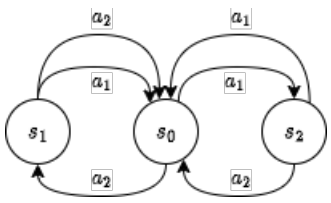
b) Given this treatment protocol/trading strategy, what is my expected return?

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 12.2.** 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 $v_\pi$ for $\pi(s) = a_1(\rightarrow), \forall s$?

  According to the Bellman Equations:

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

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

  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 $v_\pi(s_2)$:

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

  Therefore,

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

  This means that

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

  Finally,

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

  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 $v_\pi$ 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.

## 12.2. 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.

### 12.2.1. 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 12.1.** 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 |

### 12.2.2. 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* 12.1. It is possible to show that

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

**Example 12.2.** 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 $v^\pi$.
- Policy improvement: generate $\pi' \geq \pi$.

It is natural to ask if policy evaluation need to converge to $v^\pi$ 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.

## 12.3. Extensions to Dynamic Programming

### 12.3.1. 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:

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

$$v = v_{new}.$$

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

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

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

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

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 $n = |S|$.

# 13. Model-Free Prediction

## 13.1. Monte Carlo Algorithms

We can use experience samples to learn without a model. The direct sampling of episodes is called **Monte Carlo**, and this is a model-free approach, because we don't need knowledge about the MDP, only the samples.

### 13.1.1. Monte Carlo Policy Evaluation

We consider sequential decision problems, and our goal is to learn $v_\pi$ from episodes of experience under policy $\pi$:

$$S_1, A_1, R_2, ..., S_k \sim \pi.$$

The **return** is the total discounted reward, for an episode ending at time $T < t$:

$$G_t = R_{t+1} + \gamma R_{t+1} + ... + \gamma^{T-t-1} R_T.$$

The **value function** is the expected return:

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

We could also use just the sample average return, instead of the expected return.

### 13.1.2. First-Visit Monte Carlo Policy Evaluation

```
def FV_MC_PE:
  # Initialization
  foreach s:
    N(s) = 0
    G(s) = 0

  # Computation
  loop:
    # Sample episode
    e_i = s[i,1], a[i,1], r[i,1], s[i,2], a[i,2], r[i,2], ..., s[i,T_i]

    # Compute return from time step t onwards in episode i
    G[i,t] = r[i,t] + gamma * r[i,t+1] + ... + gamma ** (T_i - 1) * r[i, T_i]

    foreach time step t till the end of episode i:
      s = e_i[i, t].state
      if not visited[s, i]:
        N(s) = N(s) + 1   # Increment total visits counter
        G(s) = G(s) + G[i,t]  # Increment total return
        V_pi(s) = G(s) / N(s)   # Update estimate
```

Properties:

- $V^\pi$ is an unbiased estimator of the true $\mathbb{E}_\pi[G_t | s_t = s]$.
- By the law of large numbers, as $N(s) \to \infty$ we have $V^\pi \to \mathbb{E}_\pi[G_t | s_t = s]$.

### 13.1.3. Every-Visit Monte Carlo Policy Evaluation

```
def EV_MC_PE:
  # Initialization
  foreach s:
    N(s) = 0
    G(s) = 0

  # Computation
  loop:
    # Sample episode
    e_i = s[i,1], a[i,1], r[i,1], s[i,2], a[i,2], r[i,2], ..., s[i,T_i]
```

```
11
12      # Compute return from time step t onwards in episode i
13      G[i,t] = r[i,t] + gamma * r[i,t+1] + ... + gamma ** (T_i - 1) * r[i, T_i]
14
15      foreach time step t till the end of episode i:
16        s = e_i[i, t].state
17        N(s) = N(s) + 1   # Increment total visits counter
18        G(s) = G(s) + G[i,t]  # Increment total return
19        V_pi(s) = G(s) / N(s)   # Update estimate
```

Properties:

- $V^\pi$ is a biased estimator of $\mathbb{E}_\pi [G_t | s_t = s]$.
- But it is consistent and often with better MSE.

### 13.1.4. Incremental Monte Carlo Policy Evaluation

```
1  def EV_MC_PE:
2    # Initialization
3    foreach s:
4      N(s) = 0
5      G(s) = 0
6
7    # Computation
8    loop:
9      # Sample episode
10     e_i = s[i,1], a[i,1], r[i,1], s[i,2], a[i,2], r[i,2], ..., s[i,T_i]
11
12     # Compute return from time step t onwards in episode i
13     G[i,t] = r[i,t] + gamma * r[i,t+1] + ... + gamma ** (T_i - 1) * r[i, T_i]
14
15     for j=1:T_i:
16       V_pi(s[j,t]) = V_pi(s[j,t]) + alpha * (G[j,t] - V_pi(s[j,t]))
```

These algorithms can be used to learn value predictions, but when episodes are long, the learning process can be slow, because we have to wait until an episode ends before we can learn. In addition, the return can have high variance. Therefore, it would be nice to have other methods.

## 13.2. Temporal Difference Learning

The core of TD learning is the temporal difference error, which measures the difference between the estimated value of the current state and the estimated value of the next state, combined with the reward received for transitioning between these states. This error guides the update of the value function.

Therefore, TD is model-free and learns directly from experience. It can also learn from incomplete episodes, by bootstrapping.

TD can learn during each episode, it does not need to complete it.

### 13.2.1. Temporal Difference Learning by Sampling Bellman Equations

Recall the Bellman equations,

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

which can be approximated by iterating

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

But we can also sample this, as

$$v_{t+1}(S_t) = R_{t+1} + \gamma v_t(S_{t+1}).$$

However, this is likely to be very noisy, so it is better to take a small step,

$$v_{t+1}(S_t) = v_t(S_t) + \alpha\left(R_{t+1} + \gamma v_t(S_{t+1}) - v_t(S_t)\right).$$

The red part is the target value, and $\delta_t = R_{t+1} + \gamma v_t(S_{t+1}) - v_t(S_t)$ is called the **TD error**.

We can visualize the difference between the Dynamic Programming approach, the Monte Carlo approach and the Temporal Difference approach in the following figure:

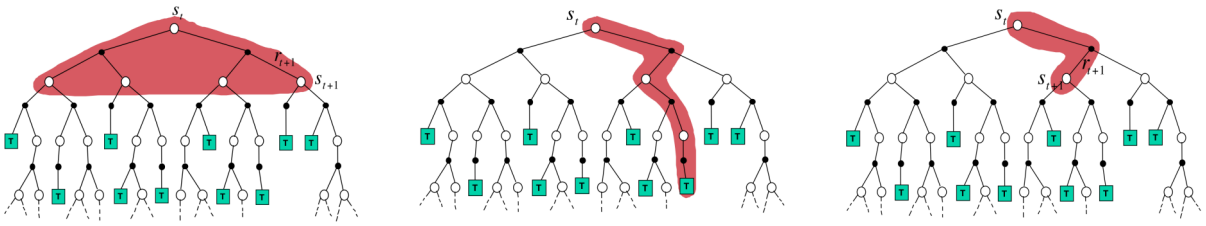


Figure 6: Backups for the different methods. DP (left), MC (center), TD (right).

In the context of reinforcement learning, **bootstrapping** refers to a method where the current estimates are updated based on other estimated values, rather than solely on actual rewards and complete trajectories. **Sampling**, on the other side, refers to learning from actual experiences or interactions with the environment.

| Method | Bootstrapping | Sampling |
|---|---|---|
| DP | Yes | No |
| MC | No | Yes |
| TD | Yes | Yes |

Note now that the idea of TD can also be applied to action values, and we can update the value $q_t(S_t, A_t)$ towards the estimated return $R_{t+1} + \gamma q(S_{t+1}, A_{t+1})$ by

$$q_{t+1}(S_t, A_t) \leftarrow q_t(S_t, A_t) + \alpha\left(R_{t+1}, \gamma q_t(S_{t+1}, A_{t+1}) - q_t(S_t, A_t)\right).$$

This approach is known as **SARSA**, since it uses $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$.

## 13.3. Comparing MC and TD

- TD can learn before knowing the final outcome, online after every step, while MC must wait until the end of the episode before the return is known.
- TD can learn without of the final outcome. It can learn from incomplete sequences and it works it continuing (non-terminating) environments. MC, on the other hand, can only learn from complete sequences, and only works for episodic (terminating) environments.

- TD is independent of the temporal span of the prediction, being able to learn from single transitions, while MC must store all predictions to update at the end of an episode.
- TD needs reasonable value estimates.
- MC returns an unbiased estimate $v_\pi(S_t)$, while TD returns a biased estimate of $v_\pi(S_t)$, but the TD target has lower variance, because the total return depends on many random actions, transitions and rewards, while the TD target depends on one random action, transition and reward.
- In some case, TD can have irreducible bias.
- When the world is partially observable, MC would implicitly account for all the latent variables.
- The function to approximate the values may fit poorly.
- In the tabular case, both MC and TD will converge to $v_\pi$.

## 13.4. Batch Monte Carlo and Temporal Difference

Tabular MC and TD converge as experience keeps going to infinity and $\alpha_t \to 0$. But what about finite experience?

Consider a fixed batch of experience:

$$\begin{aligned} &episode\ 1 : S_1^1, A_1^1, R_2^1, ..., S_{T_1}^1 \\ &\vdots \\ &episode\ K : S_1^K, A_1^K, R_2^K, ..., S_{T_1}^K \end{aligned}$$

We can repeatedly sample each episode $k \in [1, K]$ and apply MC or TD(0), which is equivalent to sampling from an empirical model.

MC converges to the best mean-square fit for the observed returns:

$$\sum_{k=1}^{K} \sum_{t=1}^{T_k} \left( G_t^k - v\left(S_t^k\right) \right)^2.$$

TD converges to solution of the maximum likelihood Markov model, given the data.

**Example 13.1.** Consider two states, $A$ and $B$, with no discounting, and the following 8 episodes of experience:

$$\begin{aligned} &A : 0, B : 0 \\ &B : 1 \\ &B : 1 \\ &B : 1 \\ &B : 1 \\ &B : 1 \\ &B : 1 \\ &B : 0 \end{aligned}$$

What are $v(A)$ and $v(B)$?

MC:

All episodes regarding $A$ have 0 reward, therefore

$$v(A) = 0.$$

Episodes regarding $B$ have 1 reward 75% of the time, so

$$v(B) = 0.75.$$

TD:

In this case, we observe that

$$p(S_{t+1} = B | S_t = A) = 1,$$

that is, whenever we go through state $A$, then we go to state $B$. This means that the reward of $A$ is the same as the reward for $B$,

$$v(A) = v(B) = 0.75.$$

TD exploits the Markov property, and so it can be helpful in fully-observable environments. On the other side, MC does not rely the Markov property, so it can be useful in partially-observable environments.

When the data is finite or we are approximating the functions, the solutions may differ.

### 13.5. Multi-Step Temporal Difference

TD uses value estimates, which can be innacurate. In addition, the information can propagate quite slowly. On the other side, the information propagates faster, but the updates are noisier. We can go in between the two methods!

The idea is by applying TD, but instead of doing just one step, we allow it to target $n$ steps into the future.

The returns, in this case, are:

$$\begin{aligned}
n = 1 \qquad & G_t^{(1)} = R_{t+1} + \gamma v(S_{t+1}) \\
n = 2 \qquad & G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 v(S_{t+2}) \\
\vdots \qquad & \vdots \\
n = \infty \qquad & G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-t-1} R_T
\end{aligned}$$

As we can see, for $n = 1$ we obtain the regular TD, and for $n = \infty$ we obtain MC.

The **learning process** is done by

$$v(S_t) \leftarrow v(S_t) + \alpha \left( G_t^{(n)} - v(S_t) \right).$$

## 14. Model-Free Control

Model-Free Control refers to learning optimal policies directly from experience (i.e., from interaction with the environment) without needing a model of the environment. In model-based control, you know the dynamics of the environment (transition probabilities and rewards), but in model-free control, you do not.

## 14.1. Monte-Carlo Control

In the previous chapter, we focused on how to estimate the value function, given a policy. Now, we are going to try to discover new policies using these estimations. Monte-Carlo control is based on estimating the value of actions through sampling. This means that the agent tries out actions and observe the outcomes, and learns from these.

When we have a model, we saw how we can use a greedy policy improvement over $v(s)$, as

$$\pi'(s) = \arg\max_a \mathbb{E}\left[R_{t+1} + \gamma v(S_{t+1}) \,|S_t = s, A_t = a\right].$$

However, when we don't have the model, we cannot use this approach. Instead, we can use a greedy policy improvement over the action-value function, as

$$\pi'(s) = \arg\max_a q_{\pi'}(s, a),$$

which makes the action-value function very convenient for this problem.

The idea, then, would be something like:

- Policy evaluation: Monte-Carlo policy evaluation, $q \sim q_\pi$. That is, from state $s_t$, we try some actions, $a_1, ..., a_k$. and estimate $q(s_t, a_1), ..., q(s_t, a_k)$.
- Policy improvement: Finding a better policy, given the value function estimated. Here, the policy is improved by making it greedy with respect to the estimated action-value function ($q$). This is:
$$a_t = \arg\max_{i=1,...,k} q(s_t, a_i).$$

But there is also a problem with this approach! We are learning by interacting with the environment, so we cannot sample all states $s$ and actions $a$: since learning is from interaction, not all state-action pairs may be visited, which makes it difficult to estimate their values accurately. This is particularly a problem in environments with a large number of states or actions.

We find a trade-off between **exploitation** (use the information we already know) and **exploration** (try new actions to discover more about the environment). One approach that increases the exploration is the **$\epsilon$-greedy approach**:

- With probability $1 - \epsilon$, select greedy action,
$$a = \arg\max_{a \in \mathcal{A}} q(s_t, a).$$
- With probability $\epsilon$, select a random action.

This can be written, for each action $a$, as

$$\pi_t(a) = \begin{cases} (1-\epsilon) + \frac{\epsilon}{|\mathcal{A}|} & if\ q(s_t, a) = \max_b q(s_t, b) \\ \frac{\epsilon}{|\mathcal{A}|} & otherwise \end{cases}.$$

This approach keeps exploring and obtaining new information.

We can write the model-free control as the following algorithm:

- Repeat:
  - Sample episode $1, ..., k, ...$ using $\pi : \{S_1, A_1, R_2, ..., S_T\} \sim \pi$.

– For each state $S_t$ and action $A_t$ in the episode, do

$$q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha_t (G_t - q(S_t, A_t)).$$

– Improve policy:

$$\epsilon \leftarrow \frac{1}{k},$$
$$\pi \leftarrow \epsilon - greedy(q).$$

Some possibilities for $\alpha_t$ are

$$\alpha_t = \frac{1}{N(S_t, A_t)},$$

the inverse of the amount of possible actions in state $t$; or

$$\alpha_t = \frac{1}{k},$$

the inverse of the amount of episodes considered.

### 14.1.1. Greedy in the Limit with Infinite Exploration (GLIE)

GLIE is a strategy to ensure that the learning algorithm both explores the environment adequately and also converges to a near-optimal policy.

*Infinite Exploration* ensures that every state-action pair is explored infinitely often. In practical terms, this means that the learning algorithm never stops trying out new actions, even if it has already found actions that seem to give good results. This continuous exploration is crucial for making sure that the algorithm doesn't miss out on potentially better actions that haven't been tried enough. Infinite exploration is often implemented using strategies like $\epsilon$-greedy. In other words, all state-actions are explored infinitely many times:

$$\lim_{t\to\infty} N_t(s, a) = \infty, \forall s, a.$$

*Greedy in the Limit* means that as the learning process continues (i.e., as the number of iterations goes to infinity), the policy becomes increasingly greedy with respect to the estimated value function. In other words, over time, the policy relies more and more on the knowledge it has gained about the environment, and less on random exploration. This is typically achieved by gradually reducing the $\epsilon$ parameter in an $\epsilon$-greedy policy. As $\epsilon$ approaches zero, the policy becomes completely greedy, always choosing the action that it currently believes to be the best. This means that

$$\lim_{t\to\infty} \pi_t(a|s) = \mathcal{I}\left(a = \arg\max_{a'} q_t(s, a')\right).$$

An example is using $\epsilon$-greedy with $\epsilon_k = \frac{1}{k}$.

**Theorem 14.1.** *GLIE Model-Free Control converges to the optimal action-value funtion,* $q_r \to q^*$.

## 14.2. Temporal-Difference Learning for Control

We saw that TD learning has several advantages over Monte-Carlo, as it shows lower variance, it learns online, and from incomplete sequences. Therefore, it is natural to think about using TD instead of MC for control. Instead of estimating $q(s,a)$ with MC, we do it with TD.

We can also update the action-value functions with SARSA, as studied in the previous section, by

$$q_{t+1}(S_t, A_t) = q_t(S_t, A_t) + \alpha_t \left(R_{t+1} + \gamma q(S_{t+1}, A_{t+1}) - q(S_t, A_t)\right).$$

Once we evaluate $q \sim q_\pi$, we improve the policy with the $\epsilon$-greedy approach.

More concretely, the following algorithm represents the approach called **tabular SARSA**:

```
initialize q(s,a) arbitrarily

for each episode:
  initialize s
  choose a from a using policy derived from q

  for each step in the episode:
    take action a
    observe r, s'
    choose a' from s' using policy derived from q

    q(s,a) <- q(s,a) + alpha * [ r + gamma * q(s',a') - q(s,a) ]

    s <- s'
    a <- a'
  until s is terminal
```

**Theorem 14.2.** *Tabular SARSA converges to the optimal aciton-value function, $q \to q^*$, if the policy is GLIE.*

Observe why it is called Tabular SARSA: we store the action-value function as a matrix of size $|\mathcal{S}| \times |\mathcal{A}|$.

## 14.3. Off-Policy Temporal-Difference and Q-Learning

**Definition 14.1. On-Policy Learning** consists on learning about the behavior policy $\pi$ from experience, sampeld from $\pi$.
**Off-Policy Learning** consists about learning the target policy $\pi$ from experiences sampled from $\mu$. It tries to learn counterfactually about other things the agent could do, by asking 'what if...?'.

**Example 14.1.** What I turned left? $\to$ New observations, rewards?

What if I played more defensively? $\to$ Do I change the win probability?

What if I continued to go forward? $\to$ How long until I bump into a wall?

The approaches seen so far are on-policy learning, while off-policy learning approaches go as follows:

- Evaluate target policy $\pi(a|s)$ to compute $v_\pi(s)$ or $q_\pi(s,a)$.

- Use a behavior policy $\mu(a|s)$ to generate actions.

Now, why is this important? It is important because it enables an agent to learn by observing humans or other agents, and to re-use experience from old policies. In addition, the agent can learn several policies, while only following one. Finally, it learns about the greedy policy while following an exploratory policy.

A common approach is **Q-Learning**, which estimates the value of the greedy policy as

$$q_{t+1}(s,a) = q_t(S_t, A_t) + \alpha_t \left( R_{t+1} + \gamma \max_{a'} q_t(S_{t+1}, a') - q_t(S_t, A_t) \right).$$

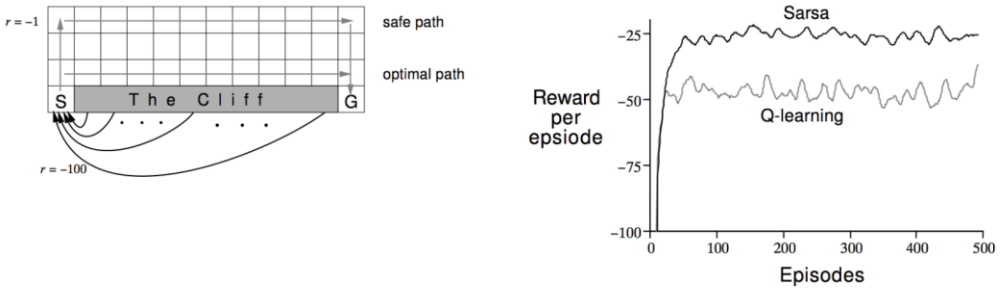
**Theorem 14.3.** *Q-Learning control converges to the optimal action-value function, $q \to q^*$, as long as we take each action in each state infinitely often.*

*Remark* 14.1. Note that Q-Learning achieves convergence without the need for greedy behavior. It works for any policy that eventually selects all actions sufficiently often.

It requires appropriately decaying step sizes, having $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t < \infty$.

One possibility is using $\alpha_t = \frac{1}{t^\omega}$, with $\omega \in (0.5, 1)$.

**Example 14.2.** In this picture we can observe how Q-Learning is more conservative than SARSA. SARSA manages to find the optimal path, but this is quite close to the cliff, and is therefore dangerous. Q-Learning finds a path that is less efficient, but is far from the cliff, minimizing the danger to fall off.




## 14.4. Overestimation in Q-Learning

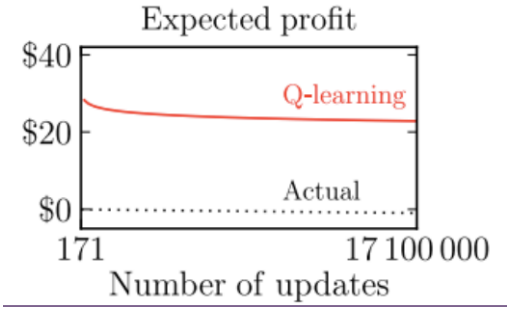
Classical Q-Learning has potential issues, due to the fact it uses the same values to select, and to evaluate; but these values are approximate, so the maximizing nature of the approach makes it more likely to select overestimated values, and less likely to select understimated values, causing **upward bias**.

**Example 14.3.** Roulette example

There are 171 actions: bet 1€ on one of 170 options, or stop.

Stops ends the episode, with 0€. All other actions have high variance reward, with negative expected value, and betting actions do not end the episode, instead enables the agent to bet again.

In the following graph, we observe how Q-Learning overestimates the expected profit in this problem:




The solution to this problem is to decouple selection from evaluation: **double Q-Learning**. The idea is to store two action-value functions, $q$ and $q'$, and update the reward estimates using them both:

$$R_{t+1} + \gamma q'_t \left(S_{t+1}, \arg\max_a q_t \left(S_{t+1}, a\right)\right),$$

$$R_{t+1} + \gamma q_t \left(S_{t+1}, \arg\max_a q'_t \left(S_{t+1}, a\right)\right).$$

At each $t$, we pick $q$ or $q'$ (randomly or with some design) and update using one of the two previous formulas, depending on which one we chose.

We can alse use both to act, for example by using the policy $\frac{q+q'}{2}$.

Double Q-Learning also converges to the optimal policy under the same conditions as Q-Learning, and solves the overestimation problem.

Moreover, this idea can be generalized to other update approaches, like **double SARSA**.

## 14.5. Importance of Sampling Corrections

Off-Policy Learning is basically trying to solve the following problem:

Given some function $f$, with random inputs $X$ and a distribution $d'$, estimate the expectation of $f(X)$ under a different distribution $d$ (the target distribution).

This can be solved by weighting the data using the ratio $\frac{d}{d'}$:

$$\mathbb{E}_{x\sim d}\left[f(x)\right] = \sum d(x) f(x) = \sum d'(x) \frac{d(x)}{d'(x)} f(x) = \mathbb{E}_{x\sim d'}\left[\frac{d(x)}{d'(x)} f(x)\right].$$

The intuition is that we scale up events that are rare under $d'$, but common under $d$, and scale down events that are common under $d'$, but rare under $d$.

**Example 14.4.** Estimate one-step reward, with behavior $\mu(a|s)$. Then

$$\mathbb{E}\left[R_{t+1}|S_t = s, A_t \sim \pi\right] = \sum_a \pi(a|s) r(s,a) = \sum \mu(a|s) \frac{\pi(a|s)}{\mu(a|s)} r(s,a) = \mathbb{E}\left[\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} R_{t+1}|S_t = s, A_t \sim \mu\right].$$

Therefore, when following policy $\mu$, we can use $\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} R_{t+1}$ as unbiased sample.

# 15. Value approximation & Deep Reinforcement Learning

## 15.1. Approximate Model-Free Prediction

Tabular RL does not scale to large, complex problems, because there are too many states to store in memory, and it is too slow to learn the values of each state separately. Therefore, we need to generalise what we learn across states.

We can estimate values or policies in an approximate way:

1. Map states $s$ onto a suitable feature representation $\phi(s)$.
2. Map features to values through a parametrised funciton $v_\theta(\phi(s))$.
3. Update parameters $\theta$ so that $v_\pi(s) \sim v_\theta(\phi(s))$.

The **goal** is then to find $\theta$ that minimises the difference between $v_\pi$ and $v_\theta$,

$$L(\theta) = \mathbb{E}_{S\sim d}\left[(v_\pi(S) - v_\theta(S))^2\right],$$

where $d$ is the state visitation distribution induced by $\pi$ and the dynamics $p$.

For this, we can use gradient descent to iteratively minimise this objective function:

$$\Delta\theta = -\frac{1}{2}\alpha\nabla_\theta L(\theta) = \alpha\mathbb{E}_{S\sim d}\left[(v_\pi(S) - v_\theta(S))\nabla_\theta v_\theta(S)\right].$$

The evaluation of the expectation is going to be a problem, and therefore the solution is to use stochastic gradient descent, sampling the gradient update:

$$\Delta\theta = \alpha(G_t - v_\theta(S_t))\nabla_\theta v_\theta(S_t),$$

where $G_t$ is a suitable sampled estimate of the return.

For Monte Carlo Prediction, it is

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ...$$

and for Temporal Difference Prediction it is

$$G_t = R_t + \gamma v_\theta(S_{t+1}).$$

At this point, it is pretty evident that we can use Deep Learning in this setting!

## 15.2. Deep Reinforcement Learning

Before, the feature representation was typically fixed, and the parametrized function was just a linear mapping. We will consider more complicated non-linear mappings $v_\theta$.

A popular choice is to use DNNs to parameterize such mapping. These are known to discover useful feature representation tailored to the specific task, and we can leverage extensive research on architectures and optimisation from DL.

Therefore, we are going to parameterize $v_\theta$ using a DNN. For instance, as a MLP:

$$v_\theta(S) = W_2 \cdot tanh(W_1 \cdot S + b_1) + b_2,$$

where $\theta = \{W_1, b_1, W_2, b_2\}$.

When $v_\theta$ is linear, $\nabla v_\theta$ is trivial to compute. However, now we have to use automatic differentiation with back-propagation implemented in DL frameworks.

**DeepMind Atari Breakthrough**

DeepMind was able to develop a Deep Reinforcement Learning system that obtained human performance in all classic Atari games. The did it by using function approximation to help scale ip to making decisions in very large domains.

To achieve such thing, they developed the concept of **Deep Q-Networks (DQN)** which consists on neural networks that are used to represents the value function, $Q$, policies and models, optimising a loss function by SGD.

Therefore, we represent the state-action value function by a DQN with weights $w$:

$$\hat{Q}(s, a; w) \approx Q(s, a),$$

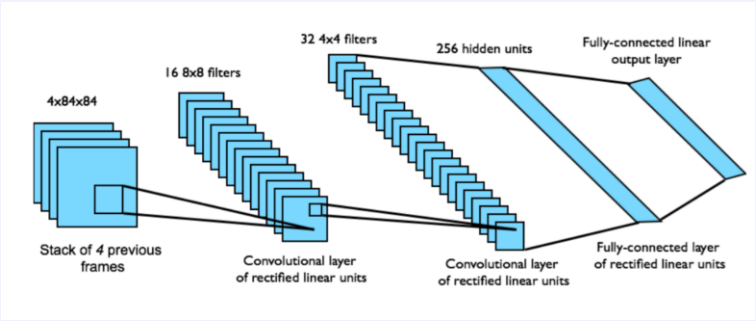
for which we:

1. Map the input state, $s$, to a $h$-dimensional vector.
2. Apply a non-linear transformation, like $tanh$ or $ReLU$.
3. Map the hidden vector to the $m$ possible action values, as a classification problem.
4. The action with greatest approximated $Q$-value is selected.

It is also possible to pass each action to the network, and output its $Q$-value. This is useful when there are too many actions or the action space is continuous.

**DeepMind Atari Breakthrough**

The performed end-to-end learning of the values $Q(s, a)$ from the states, $s$, represented by the pixels of the game of the last 4 frames of the game.
The output is the value $Q(s, a)$ for the 18 possible button positions.
The reward is a change in the score for that step.
The network architecture and hyperparameters were the same for all the games of the console!




We saw in previous sections that $Q$-learning converges to the optimal $Q^*(s, a)$ using table lookup representation. When working with value function approximation, we can minimize the MSE loss by SGD using a target $Q$ estimate instead of the true $Q$, like in linear value function approximation.

However, $Q$-learning with value function approximation can diverge from the optimal $Q^*$, due to mainly the correlations between samples, and the existence of non-stationary targets.

DQN addresses these challenges by leveraging **experience replay** and **fixed $Q$-targets**.

### 15.2.1. Experience Replay

To help remove correlations, we can store a dataset, called the **replay buffer**, $\mathcal{D}$, obtained from prior experience. In this dataset, we store at each step $i$, the tuple $(s_i, a_i, r_{i+1}, s_{i+1})$, that is:

- The initial state.
- The action taken.
- The reward.
- The obtained state.

Once we have the dataset, we would perform experience replay as follows:

1. Sample an experience tuple from the replay buffer:.
$$(s, a, r, s') \sim \mathcal{D}.$$
2. Compute the target value for the sampled $s$ as $r + \gamma \max_{a'} \hat{Q}(s', a'; w)$.
3. Use SGD to update the network weights:
$$\Delta w = \alpha \left( r + \gamma \max_{a'} \hat{Q}(s', a'; w) - \hat{Q}(s, a; w) \right) \nabla_w \hat{Q}(s, a; w),$$
obtained from the loss function
$$L(w) = \frac{1}{2} \left( r + \gamma \max_{a'} \hat{Q}(s', a'; w) - \hat{Q}(s, a; w) \right)^2.$$

## 15.3. Deep Learning aware Reinforcement Learning

As always, things are not all good, and there exist issues with online deep RL. We know from DL literature that

- SGD assumes gradients are sampled IID.
- Using mini-batches instead of single samples is typically better.

However, in online RL, we perform an update on every new sample, and consecutive updates are strongly correlated.

We can be smart in the design of the method to improve it. For instance, Experience Replay helps with this, because we can mix online updates with updates on data sampled from the replay buffer. This approach can reduce correlation between consecutive updates, and enables mini-batch updates, instead of vanilla SGD.

Other ways to address this issue are:

- Use better online algorithms, like eligibility traces.
- Use better optimizers, like with momentum.
- Planning with learnt models, like Dyna-Q.
- Changing the problem setting itself, like using parallel environments.

### 15.3.1. The Deadly Triad

If we use experience replay, we are combining:

1. **Function approximation**: with the mere use of a neural network to fit the action values.
2. **Bootstrapping**: because we bootstrap on $\max_a Q_\theta(s,a)$ to construct the target.
3. **Off-policy learning**: the replay hold data from a mixture of past policies.

When these three elements are combined, they can create a feedback loop where errors in the function approximation are compounded through bootstrapping, and the off-policy nature of the learning algorithm can lead to poor or incorrect updates. This can cause the learning algorithm to diverge, meaning that instead of converging to a solution, it becomes increasingly unstable over time.

Empirically, it is actually rare to see unbounded divergence. The most common scenarios are value explosions that recover after an initial phase, a phenomenon referred to as **soft-divergence**.

Soft-divergence still cause value estimates to be quite poor for extended periods... But we can address this in our RL agents using a separate **target network**, that is:

- Hold fixed the parameters used to compute the bootstrap targets $\max_a Q_\theta(s,a)$.
- Only update them periodically (every few hundreds or thousands of updates).

This way, we can break the feedback loop that sits at the heart of the deadly triad.

The general pseudocode of DQN is the following:

```
def DQN(C, alpha, gamma):
  D = {} #Replay buffer
  Initialize w #DQN
  w_ = w #Target network
  t = 0
  Get initial state s_0

  while True:
    sample action a_t given eps-greedy policy for current Q'(s_t,a;w)
    Observe reward r_t
    Observe next state s_t+1
    D.append(s_t,a_t,r_t,s_t+1)
    sample random mini-batch of tuples (s_i,a_i,r_i,s_i+1) from D

    for j in mini-batch do
      if episode terminated at step i+1 then
        y_i = r_i
      else
        y_i = r_i + gamma * max_a[Q'(s_i,a_i;w_)]

      do SGD step on (y_i - Q'(s_i,a_i;w))^2
      update w += dw
    t++
    if t % C == 0:
      w_ = w
```

### 15.3.2. Deep Double Q-Learning

We've said that Q-Learning has an overstimation bias, and this can be correcpted by double Q-Learning. That is, we have two models: the current network, $q_\theta$, and the target network, $q_{\theta^-}$. The

current network is used to select the best action, and the target network is used to evaluate the Q-value of that action. By decoupling the action selection from its evaluation (using different networks), DDQN mitigates the overestimation bias. When updating the Q-value for a state-action pair, the current network determines the action that maximizes the Q-value, but the target network provides the Q-value for that action. Periodically, the weights of the target network are updated to match those of the current network. This lag in updating helps stabilize learning. The likelihood function in this case becomes:

$$L(\theta) = \frac{1}{2}\left(R_{i+1} + \gamma \cdot q_{\theta^-}\left(S_{i+1}, \arg\max_a q_\theta(S_{i+1}.a)\right) - q_\theta(S_i, A_i)\right)^2.$$

### 15.3.3. Prioritized Replay

DQN samples uniformly from replays, so an idea would be to prioritize those transitions with which the model can learn more. A basic implementation is to set a priority for sample $i$, $|\delta_i|$, as the TD error on the last time this transition was sampled. Then, we sample according to this priority.

### 15.3.4. Multi-Step Control

In the context of Deep Q-Learning (DQL), multi-step control is integrated to enhance the learning process by considering a sequence of actions and their cumulative rewards over multiple steps, rather than relying on single-step rewards and transitions. We define the $n$-step Q-Learning target as

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+1} + ... + \gamma^{n-1} R_{t+n} + \gamma^n q_{\theta^-}\left(S_{i+1}, \arg\max_a q_\theta(S_{i+1}, a)\right),$$

with the update rule

$$\Delta\theta = \alpha\left(G_t^{(n)} - q_\theta(S_t, A_t)\right)\nabla_\theta q_\theta(S_t, A_t).$$

The return is partially on-policy, while the bootstrap is off-policy: For a certain number of steps ($n$ steps), the agent follows its current policy (on-policy learning). It accumulates rewards for these steps and uses them to update its value estimates. After these $n$ steps, the agent uses a value estimate from a target policy (which could be a greedy policy) for further updates. This part is off-policy because it involves using a different policy (the target or greedy policy) for the value update.

A well-defined target: "On-policy for $n$ steps, then act greedy".

That's, okay: less greedy, but still a policy improvement. This method allows the agent to explore and learn effectively from its own policy while still benefiting from the stability and efficiency of using a greedy policy for value estimation.

## 15.4. Reinforcement Learning Aware Deep Learning

### 15.4.1. Architectures

Much of the successes of DL have come from encoding the right inductive bias in the network structure. To name a few examples:

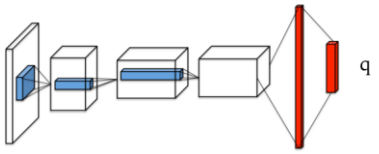
- Translation invariance in image recognition led to convolutional networks.
- Long term memory led to gating in LSTMs.
- Permutation invariance/equivariance led to attention in transformers.

We should not just copy architectures designed for supervised problems, but rather design the appropriate architecture to encode inductive biases that are good for RL.

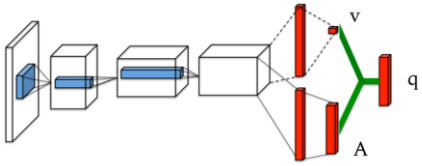
**Dueling networks** We can decompose

$$q_\theta(s,a) = v_\xi(s) + A_\chi(s,a),$$

where $\theta = \xi \cup \chi$, with $A_\chi$ representing the adcantage of taking action $a$. That is, we modify




to be:




The Dueling Network architecture splits the deep neural network into two distinct pathways at its final layers:

- One for estimating the state value function ($V(s)$), which gives the value of being in a particular state.
- The other for estimating the action advantage function ($A(s,a)$), which represents the additional value of taking a particular action in that state, compared to other actions.

The outputs of these two pathways are then combined to estimate the Q-value function. This is typically done by adding the state value to the action advantage, but with an adjustment to maintain a zero advantage at the chosen action. The Q-value function represents the value of taking a certain action in a given state.

The final output is a set of Q-values for each action in the given state, just like in a traditional DQN. However, the way these Q-values are computed is different due to the dueling structure.

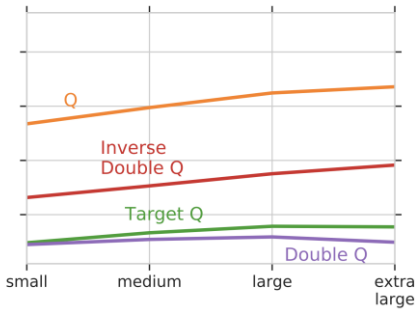
The advantages of this architecture over traditional ones are:

1. Improved Learning of State Values: Since one pathway is dedicated to learning the state value function, the network can more accurately learn the value of each state, regardless of the actions available.
2. Efficient Learning of Action Advantages: By separately estimating the advantage of each action, the network can more effectively distinguish between the importance of the state and the importance of the individual actions within that state. This leads to more nuanced policy learning.
3. Stability and Robustness: Dueling networks can lead to more stable learning and improved convergence properties. By disentangling the estimation of state values and action advantages, the network reduces the potential for harmful interference between these two aspects of the learning process.
4. Better Policy Evaluation and Generalization: The architecture allows for better generalization across actions without compromising the learning of the state value. It can also provide a more accurate evaluation of the policy.

5. Handling Environments with Many Similar Actions: In situations where many actions yield similar results (e.g., in video games where moving left or right might have similar outcomes), a dueling network can effectively learn the value of the state, which is more significant than the specific action taken.

### 15.4.2. Capacity

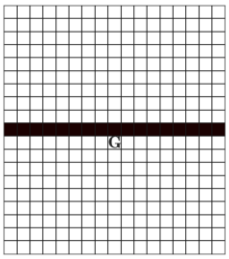
In supervised DL, we often find that the more data and compute we have, the better performance we can get, since the loss is easier to optimise, there is less interference, we can have bigger models, etc. But this phenomenon is not found the same way in reinforcement learning, where larger network do typically perform better overall, but become more susceptible to the deadly triad.



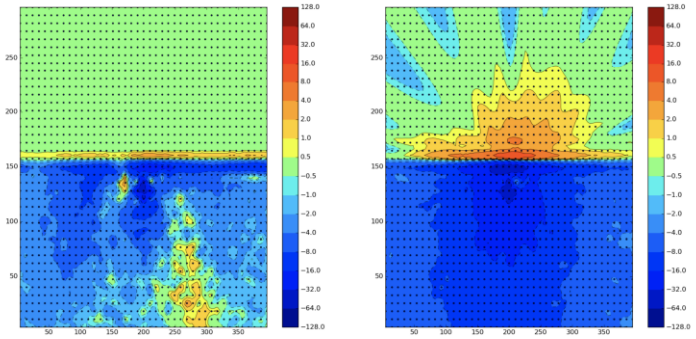
(b) Network size

### 15.4.3. Generalisation

The deadly triad also shows that generalization in RL can be tricky... Consider the problem of value learning, in presence of sharp discontinuities of $v_\pi$. Deep neural networks, used in deep reinforcement learning (like DQN or policy gradient methods), might struggle to approximate functions with sharp discontinuities accurately. This is because neural networks, by their nature, tend to interpolate smoothly between points in their training data. For example:




TD learning with DL approximation leads to what's called **leakage propagation**, that is, the discontinuities spread their values to undesired areas of the space:



(c) MC prediction error heatmap (d) TD prediction error heatmap

## References

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

[2] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks.