UNIVERSIDAD
DE GRANADA

# MUTUAL INFORMATION IN SELF-SUPERVISED LEARNING

FRANCISCO JAVIER SÁEZ MALDONADO

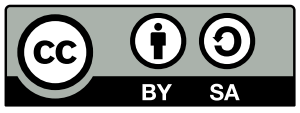
Bachelor's Thesis

Computer Science and Mathematics

**Tutor**
Nicolás Pérez de la Blanca Capilla

FACULTY OF SCIENCE
H.T.S. OF COMPUTER ENGINEER AND TELECOMMUNICATIONS

*Granada, Friday 3rd September, 2021*




The source code of this text and developed programs are available in the Github repository fjsaezm/Mutual-Information-in-Unsupervised-Machine-Learning

# ABSTRACT

*Representation learning* is the process of, given an input data of any type, training a model that learns to produce a lower dimensional vector that summarizes the information contained in the input. This document will focus on the main used method for representation learning.

Firstly, an introduction to the basic probability concepts is made, along with an introduction on the contrastive noise estimation problem, which will be key in the construction of the *InfoNCE* loss and great inspiration for the contrastive methods that currently achieve the state of art results in this field.

In this context,*mutual information* appears as a good method to obtain good representations by maximizing this function between the input and its representation. We review the most important properties of this measure and, since it is hard to explicitly compute it, we present some lower bounds on this function which can make the task easier.

Lastly, it was empirically shown that maximizing mutual information is not the best way of obtaining good representations for downstream tasks. New frameworks (SimCLR, BYOL) that present different perspectives of the same input to the models have shown to perform better, so we present these frameworks and review what choices have to be explored in order to achieve the best performance using these kind of methods.

**Keywords:** *representation learning, noise contrastive estimation, entropy, mutual information, lower bounds, siamese networks, triplet loss* and *deep learning*.

# RESUMEN EXTENDIDO EN ESPAÑOL

En este trabajo, trataremos de exponer cómo ha ido evolucionando el *aprendizaje de representaciones* desde los primeros métodos usados hasta los nuevos marcos usados en este campo. En este trabajo, realizaremos un estudio **teórico** de los conceptos necesarios para aproximarnos al problema de aprendizaje de representaciones (partes 1 a 3) y, a continuación y de forma algo más **práctica**, presentaremos algunos marcos de trabajo utilizados y realizaremos una serie de experimentos utilizando estos marcos (partes 4,5).

La parte **teórica** describe y profundiza los conceptos más importantes y en los que se fundamenta la parte prática.

En la **primera parte** se comienza haciendo una introducción y motivación profunda al tema (capítulo 1), y se sigue (capítulo 2) describiendo los conceptos básicos de teoría de probabilidad que se van a usar para exponer la teoría del trabajo, como el concepto de variable aleatoria, independencia o esperanza de una variable aleatoria. A continuación (capítulo 3), se da la definición de la *divergencia de Kullback-Leibler*, que será muy relevante pues será una de las formas que tengamos de expresar la *Información Mutua*. Además, se darán algunos ejemplos de distribuciones de probabilidad que son usadas en el trabajo.

Se realiza también (capítulo 4) una introducción a la inferencia estadística, introduciendo el concepto de verosimilitud y logaritmo de la función de verosimilitud. Por último (capítulo 5), se enuncia el problema de la *extimación de ruido contrastiva*, también muy relevante pues sobre él se definirán los marcos de trabajo del aprendizaje de representaciones.

La **segunda parte** está dedicada a presentar los conceptos más importantes de la *teoría de la información* en los que se basa nuestro trabajo. Primeramente (capítulo 6), se expone el concepto de *entropía* y se dan ciertas propiedades de la misma. Estas propiedades son útiles pues cuando definimos (capítulo 7) la *información mutua*, al definirse esta en función de la entropía, se pueden extrapolar sin ningún esfuerzo a propiedades de la información mutua. Además,también introducimos las tres principales *cotas inferiores* para la información mutua: la *cota inferior variacional*, la *cota inferior contrastiva* y la cota inferior usando la*representación Donsker-Varadhan* de divergencia de Kullback-Leibler. Estas cotas son muy útiles a la hora de hacer aproximaciones al valor real de la información mutua.

En la **tercera parte** se aproxima la formulación matemática al problema de *aprendizaje de representaciones* (capítulo 8). Se presenta también de forma precisa el problema que queremos abordar. A continuación, se hace una introducción a los *modelos generativos* (capítulo 9), modelos que tratan de obtener la distribución de probabilidad $P$ que sigue un conjunto de datos dado. Se además una introducción a los modelos auto-regresivos, que utilizan datos

en ciertos instantes de tiempo para predecir valores futuros de esos datos. Más adelante (capítulo 10), se introduce el marco de trabajo del aprendizaje contrastivo en el que, fijado un conjunto de datos, se trata de discriminar entre datos obtenidos de dos distribuciones de probabilidad $P$ y $Q$ diferentes. Para ello, se utiliza la función de pérdida contrastiva. Por último en esta parte, se introducen las funciones de pérdida utilizando tripletas (capítulo 11), que son una generalización de la función de pérdida contrastiva.

En la **parte práctica**, se explican los principales marcos de trabajo que se utilizan y se exponen los experimentos realizados y resultados obtenidos.

La **cuarta parte** comienza haciendo una introdución al aprendizaje profundo (capítulo 12) resaltando el *aumento de datos*, que proporciona nuevos ejemplos a partir de los datos obtenidos y que juega un papel crucial en el aprendizaje de representaciones. Entonces, se presentan dos redes siamesas:*SimCLR* (capítulo 13) y *Bootstrap your own latent (BYOL)* (capítulo 14), los dos marcos de trabajo que han surgido en el año 2020 para el aprendizaje de representaciones. En ambos casos, se da una motivación de por qué surgen, se explica la arquitectura que ambas siguen y las funciones de pérdida que utilizan cada una, y se comenta qué hiperparámetros pueden ser más relevantes a la hora de entrenar los modelos y obtener mejores representaciones para las tareas posteriores como clasificación o regresión.

En la **quinta y última parte** se exponen los experimentos realizados utilizando los marcos anteriores. Primeramente (capítulo 15), se exponen los objetivos que se persiguen mediante estos experimentos, que se resumen en adaptar los experimentos existentes a los recursos de los que se disponen. Además, se exponen las tecnologías utilizadas. Seguidamente (capítulo 16), se exponen primero los tres experimentos realizados utilizando SimCLR, un primero general, un segundo aumentando el tamaño y profundidad del *encoder* del marco, y un tercero añadiendo una nueva capa de aumento de datos y preprocesamiento de los mismos. Los experimentos resultan exitosos, obteniendo en el tercero mejoras respecto al primero y resultados acordes a lo previsto. Lo mismo oscurre más tarde cuando realizamos dos experimentos utilizando BYOL, un primero en el que se demuestra empíricamente que la influencia que tenía el *tamaño del batch* en SimCLR se pierde en BYOL, y un segundo en el que se amplía de nuevo el tamaño del encoder, obteniendo también resultados exitosos.

**Palabras clave:** *información mutua*, *estimación contrastiva de ruido*, *entropía*, *aprendizaje de representaciones*, *redes siamesas*, *pérdida usando tripletas*, *aprendizaje profundo*, *cotas inferiores*.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# Part I

# INTRODUCTION AND BASIC NOTIONS

# 1 INTRODUCTION, MOTIVATION AND OBJECTIVES

*Machine learning* is the field of computer science that studies algorithms that improve automatically through experience from examples. These algorithms allow computers to discover how to perform tasks without being explicitly programmed to do so. For the computers to learn, it is mandatory that a finite set of data (or dataset) $\mathcal{D}$ is available.

This data can be *labeled* or *unlabeled*. Labeled comes in pairs $(x_i, y_i)$, where $x_i$ is a datapoint and $y_i \in Y$ is a specific label that has previously been assigned to $x_i$. Usually, $Y$ is a set of classes or real values. Unlabeled data is the kind of data that does not have a label or class associated to it, so it is just $x_i \in \mathbb{R}^d$.

Depending on how the available data the machine learning approaches can be divided into two broad categories:

1. *Supervised learning*. In this category the goal is to use the labeled data in order to find a function that maps the dataset to the set of classes or real values. That is a function $g : \mathcal{D} \to Y$. An example of supervised learning is image classification: giving a label to an image.
2. *Unsupervised learning*. In ths case, the data is unlabeled, so the approach is completely different. Usually, the goal here is to discover hidden patterns in data or to learn features from it. An example of this kind of learning is K means, which consists of clustering the data in $k$ groups. It is also known as *self-supervised* learning.

When machine learning was born (around the 1950s), all the problems were related to supervised learning. Then, in the 1970s, *backpropagation* was developed allowing models to adapt to new situations much faster. Since then, diverse algorithms that manage to find truly complex $f$ functions have been developed and new problems have emerged. Supervised learning has a huge cost (economic, time): all the examples in the dataset must be labeled. This is so expensive due to the fact that labeling examples is a slow process, and has to be done mostly manually.

Also, a worrying generalization problem was discovered in a very simple classification problem: discriminating between cows and camels images in different environments (Beery *et al.*, 2018). The original data had a lack of different environments, that is, all the caws had grass in the background and all camels had a desert in the background. The models obtained really good results in images where the context (in this case, the background) was the same as the one we trained with. However, when we changed the context, eg: placed a cow in the snow instead of in the grass, the performance of the models drastically decreased. This led to the discovery that the models were

simply using the context to discriminate from the different animals rather than focusing on the important part of the image, the animal.

*Unsupervised learning* avoids these problems by trying to infer properties (or features) of the data using the *unlabeled* dataset. By not needing to have the labels of the examples, companies can save money and time that they would have invested creating a label for each individual example. Also, by discovering general features about the context the models, the general context of the data is less important than it sometimes happens to be in supervised learning.

When the dimension of the data is high, for instance when treating images computationally, it is usual to first create a *representation* of the input data. A representation is a lower dimensional vector that aims to contain the same features that the original data contained. More formally, if $x$ is a datapoint in a dataset $\mathcal{D} \subset \mathbb{R}^d$, a *representation* of $x$ is a vector $r \in \mathbb{R}^n$ (usually, $n \leq d$), that shares information with the datapoint $x$.

*Features* are parts or patterns of an datapoint $x \in \mathcal{D}$ that help to identify it. In fact, it is desirable that this attribute is shared by all independent units that represent the same object. For instance, if we consider an image of any square, we should be able to identify 4 corners and 4 edges. These could be features of a square.

Representation learning is a set of techniques that allow a system to produce representations that help in the later tasks such as feature detection or classification. The performance of machine learning methods is heavily dependent on the choice of data features (Bengio *et al.*, 2014). This is why most of the current effort in machine learning focuses on designing preprocessing and data transformation that lead to good quality representations. A representation will be of good quality when its features produce good results when we evaluate the *accuracy* of our model.

The main goal in representation learning is to obtain features of the data that are generally good for either of the supervised tasks. These tasks are usually called *downstream tasks*. That is, we would like to obtain a representation that is either good for image classification (giving an image a label of what we can see in it) or image captioning (producing a text that describes the image).

The features of the data that are invariant through time are very useful for machine learning models. In (Wiskott & Sejnowski, 2002), *slow features* are presented. Slow features are defined as features of a signal (which can be the input of a model) that vary slowly during time. That means, if $\boldsymbol{X}$ is a *time series*[1], we will try to find any number of features in $\boldsymbol{X}$ that vary the most slowly. These kind of features are the most interesting ones when creating representations, since they give an abstract view of the original data.

*Example* 1. In computer vision, the value of the pixels in an image can vary fast. For instance, if we have a zebra on a video and the zebra is moving from

1 A time series is an ordered sequence of values of a random variable at, usually, equally spaced time intervals.

one side of the image to the other, due to the black stripes of this animal, the pixels will fast change from black to white and vice versa, so value of pixels is probably not a good feature to choose as an slow feature. However, there will always be a zebra on the image, so the feature that indicates that there is a zebra on the image will stay positive throughout all the video, so we can say that this is a slow feature.

We will be studying different models that try to learn representations from raw data without labels, as we have mentioned. We usually need a function that measures what is the penalty that the model gets for a choice of a parameter. This is called a *loss function*, that we will want to optimize.

In the search for a suitable loss function, a first approach to the problem was to use concepts from information theory. Ideally, we would like the original data and the representation created to contain the same information. A way of measuring this is using the *mutual information* $I(X, Z)$ between the input $X$ and the representation $Z$. The mutual information is expressed as:

$$I(X, Z) = H(X) - H(X|Z),$$

where $H(X)$ is the entropy of $X$ and $H(X|Z)$ is the conditional entropy.

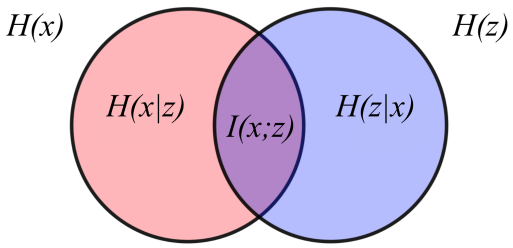



**Figure 1:** Venn Diagram showing the relationships between the entropies and the conditional entropies of random variables $X$ and $Z$.

Since the entropy is a way of measuring "*how surprising is that an event occurs*" (this is explained in this document), intuitively the mutual information measures the decrease of uncertainty that we obtain in $X$ when we know that $Z$ has occurred.

Calculating the mutual information between two variables, however, is not an computationally easy problem. Because of this, a way to approach the mutual information maximization is by obtaining lower bounds and maximizing them. Although a few more bounds are proved in this document, we remark the *Contrastive Lower Bound* on the mutual information, which is expressed as follows:

$$I(X, Z) \geq -\ell(\theta) + \log N,$$

where $\ell(\theta)$ refers to the contrastive loss. In short, the *contrastive learning* problem consists of, given elements obtained from one distribution $P$ (considered as positive samples) and elements obtained from a different distribution $Q$

(considered as negative samples), learning how to discriminate between the elements of the different distributions. In order to do this, the recently mentioned contrastive loss is used, which usually takes the form

$$\ell_{i,j} = -\log \frac{f(x_i, x_j)}{\sum_{x_k \in X} f(x_i, x_k)},$$

where $(x_i, x_j)$ is a positive pair. Intuitively, contrastive loss takes the output of the network for a positive example and calculates its distance (using the function $f$) to an example of the same class and contrasts that with the distance to negative examples. In other words, the loss is low if positive examples are encoded to similar representations and negative examples are encoded to different representations. The use of this contrastive loss maximizing the mutual information had a huge impact on the state of art results of representation learning in 2018 (Oord *et al.*, 2019), achieving very promising results.

However, during the development of this work (around July 2020), this drastically changed. New papers (Chen *et al.*, 2020b; Grill *et al.*, 2020) appeared stating that the success of the methods that were maximizing mutual information between the input and its representation was not caused by mutual information, but by the specific form that the contrastive loss has.

With this papers, new frameworks for representation learning appeared. These frameworks provided an ingenious way to apply the contrastive learning set up to representation learning. This technique consists of presenting different perspectives of the same image to two different encoders (the neural networks that learns how to transform the input to a representation) different *perspectives* (or, being technical, *data augmentations*) of the same image and, later, trying to minimize some sort of distance between these views.

To achieve this, these perspectives are obtained by applying transformations to the original image. It is shown that the chosen transformations and how to sequence them before passing the transformed image to the neural network really affects the performance of the models.

All this considered, the interest of this work speaks on its own. Being able to train networks that learn representations that are good enough on its own to perform downstream tasks without needing the supervision of having the labels of the data opens a world of possibilities in many areas. For instance, many insurance companies need to pay their workers to label large amounts of different insurance bills that they offer to their clients. With the advances presented in this work, these companies would only have to label a very small amount of images to be able to classify the representations obtained by the frameworks.

## MAIN GOALS AND RESULTS ACHIEVED

The main goals of this bachelor's thesis were:

1. To study, understand and present the basic concepts and properties of *mutual information*, which is in the field of *Information Theory.* Also,

related to this, to present some lower bounds that can be used to maximize the mutual information between two random variables.
2. To study the *noise contrastive estimation* problem, and how it is applied to machine learning.
3. To test the most important frameworks that achieve the current state of art results in representation learning.

The first two goals were successful. For the latter goal, two different frameworks were tested: `SimCLR` and `BYOL`.

Different executions were made for each one, trying to find the set of parameters for being able to obtain a high classification accuracy on a specific dataset. The selection of the chosen hyperparameters was argued and the limitations and problems we face are also presented. This way, I gained not only a deep understanding of the representation learning problem and how it is solved, but also in computational capability of the used hardware, so that a plausible future plan could be to improve the results obtained by making use of even better computational resources.

# 2 FUNDAMENTALS

Underneath each experiment involving any grade of uncertainty there is a *random variable*. This is no more than a *measurable* function between two *measurable spaces*. A probability space is composed by three elements: $(\Omega, \mathscr{A}, \mathcal{P})$. We will define those concepts one by one.

## 2.1 PROBABILITY SPACES

**Definition 2.1.1.** A *measure space* is the tuple $(\Omega, \mathscr{A}, \mathcal{P})$ where $\mathcal{P}$ is a *measure* on $(\Omega, \mathscr{A})$. If $\mathcal{P}$ is a *probability measure* $(\Omega, \mathscr{A}, \mathcal{P})$ will be called a *probability space*.

Throughout this work, we will be always in the case where $\mathcal{P}$ is a probability measure, so we will always be talking about probability spaces and we will note $\mathcal{P}$ simply as $P$. Some notation for these measures must be introduced. Let $A$ and $B$ be two events. The notation $P(A, B)$ refers to the probability of the intersection of the events $A$ and $B$, that is: $P(A, B) := P(A \cap B)$. It is clear that since $A \cap B = B \cap A$, then $P(A, B) = P(B, A)$. The following theorem will be used multiple times in the proofs that we will do in the next chapters:

**Theorem 2.1.1** (Bayes' Theorem)**.** *Let $A, B$ be two events in $\Omega$, given that $P(B) \neq 0$. Then*

$$P(B|A) = \frac{P(A|B)P(A)}{P(B)}.$$

*Proof.* Straight from the definition of the conditional probability we obtain that:

$$P(A, B) = P(A|B)P(B).$$

We also see from the definition that

$$P(B, A) = P(B|A)P(A).$$

Hence, since $P(A, B) = P(B, A)$,

$$P(A|B)P(B) = P(B|A)P(A) \implies P(A|B) = \frac{P(B|A)P(A)}{P(B)}.$$

□

*Random variables* (RV) can now be introduced, since they are one of the concepts that will lead us to the main objective of this thesis.

**Definition 2.1.2.** Let $(\Omega, \mathscr{A}, \mathcal{P})$ be a probability space, and $(E, \mathcal{B})$ be a measurable space. A *random variable* is a measurable function $X : \Omega \to E$,

from the probability space to the measurable space. This means: for every subset $B \in (E, \mathcal{B})$, its pre-image

$$X^{-1}(B) = \{\omega : X(\omega) \in B\} \in \mathscr{A}.$$

### Expectation of a random variable

Imagine observing a wide number of outcomes from our random variable, and taking the average of these random values. The expectation is the value of this average when we take *infinite* outcomes of our random variable.

**Definition 2.1.3.** Let $X$ be a non negative random variable on a probability space $(\Omega, \mathscr{A}, \mathcal{P})$. The *expectation* $E[X]$ of $X$ is defined as:

$$E[X] = \int_{\Omega} X(\omega)\, dP(\omega).$$

Sometimes we might be referring to multiple random variables. In these cases, in order to make reference to the variable (or distribution function, that will be presented later) for which we calculate the expectation, we will denote it as $E_X$ (or $E_P$, in the case that we are addressing a distribution).

Recall that the expectation $E[X]$ of a random variable is a linear operation. That is, if $Y$ is another random variable, and $\alpha, \beta \in \mathbb{R}$, then

$$E[\alpha X + \beta \mathcal{Y}] = \alpha E[X] + \beta E[\mathcal{Y}].$$

This is a trivial consequence of the linearity of the *Lebesgue integral*.

### Random vectors

Usually, when it comes to applying these concepts to a real problem, we will be observing multiple features that a phenomenon in nature presents. We would like to have a collection of random variables each one representing one of this features. In order to set the notation for these kinds of situations, we will introduce *random vectors*.

**Definition 2.1.4.** A random vector is a row vector $\mathbf{X} = (X_1, \ldots, X_n)$ whose components are real-valued random variables on the same probability space $(\Omega, \mathscr{A}, P)$.

We can also extend the notion of expectation to a random vector. Let $\mathbf{X} = (X_1, \ldots, X_n)$ be a random vector and assume that $E[X_i]$ exists for all $i \in \{1, \ldots, n\}$. The expectation of $\mathbf{X}$ is defined as the vector containing the expectations of each individual random vector, that is:

$$E[\mathbf{X}] = \begin{bmatrix} E[X_1] \\ \vdots \\ E[X_n] \end{bmatrix}.$$

It can also happen that, given a random vector, we would like to know the probability distribution of some of its components. That is called the *marginal distribution*.

**Definition 2.1.5.** Let $\mathbf{X} = (X_1, \ldots, X_n)$ be a random vector. The *marginal distribution* of a subset of $\mathbf{X}$ is the probability distribution of the variables contained in the subset.

In the simple case of having two random variables, e.g. $X = (X_1, X_2)$, then the marginal distribution of $X_1$ is:

$$P(x) = \int_{x_2} P(x_1, x_2) dx_2.$$

## 2.2 DISTRIBUTIONS AND KULLBACK–LEIBLER DIVERGENCE

We have introduced the concepts of *random variable*, *random vector* and its *probability distribution*. We will explain some concepts related to the latter ones.

**Definition 2.1.** The *mode* of a distribution is the value at which the probability mass function takes its maximum value. That is, the value that is most likely to be sampled.

Distributions can be *unimodal*, when their distribution has a single peak, *bimodal* when their distribution has two peaks, and *multimodal* when the number of peaks is equal or greater to 2.

*Example* 2. We can simulate the two following distributions:

1. The distribution of the marks obtained in a test by the students of certain class.
2. The distribution of the height of the plants from three different species.

The result is the following:

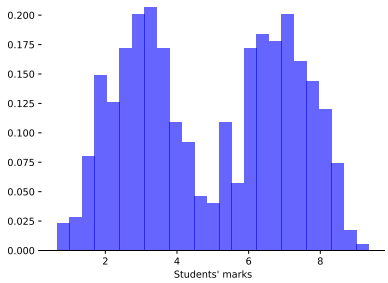



**Figure 2:** Bimodal Distribution

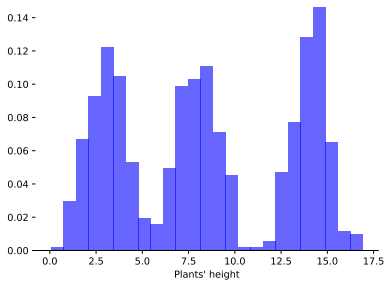



**Figure 3:** Plants' height

**Figure 4:** Examples of bimodal and multimodal distributions.

Now, given two distributions, we would like to determine how different they are from each other. In order to compare them, we enunciate the definition of the Kullback-Leibler divergence.

**Definition 2.2.** Let $P$ and $Q$ be probability distributions over the same probability space $\Omega$. Then, the Kullback-Leibler divergence is defined as:

$$D_{KL}(P \mid\mid Q) = E_P\left[\log \frac{P(x)}{Q(x)}\right].$$

It is defined if, and only if, $P$ is *absolutely continuous with respect to* $Q$, that is, if $P(A) = 0$ for any $A$ subset of $\Omega$ where $Q(A) = 0$. There are some properties of this definition that must be stated.

**Proposition 1.** *If $P, Q$ are two probability distributions over the same probability space, then $D_{KL}(P|Q) \geq 0$.*

*Proof.* Firstly, note that if $a \in \mathbb{R}^+$, then $\log\ a \leq a - 1$. Then:

$$\begin{aligned}
-D_{KL}(P \mid\mid Q) &= -E_P\left[\log \frac{P(x)}{Q(x)}\right] \\
&= E_P\left[\log \frac{Q(x)}{P(x)}\right] \\
&\leq E_P\left[\left(\frac{Q(x)}{P(x)} - 1\right)\right] \\
&= \int P(x)\frac{Q(x)}{P(x)}\,dx - 1 \\
&= 0.
\end{aligned}$$

So we have obtained that $-D_{KL}(P \mid\mid Q) \leq 0$, which implies that $D_{KL}(P \mid\mid Q) \geq 0$. □

As a corollary of this proposition, we can affirm that $D_{KL}(P \mid\mid Q)$ equals zero if and only if $P = Q$ almost everywhere. We will also remark the discrete case, as it will be used later. Let $P, Q$ be discrete probability distributions defined on the same probability space $\Omega$. Then,

$$D_{KL}(P \mid\mid Q) = \sum_{x \in \Omega} P(x) \log\left(\frac{P(x)}{Q(x)}\right).$$

### 2.2.1 Examples of distributions

Let us present some examples o common distributions. They will be used further in this document.

#### *Bernoulli*

Think for a moment that you want to model the possible outcomes of an experiment with two possibilites: success or failure. Imagine also that you already know that in your experiment there is a probability $p$ of achieving success. That is the intuitive idea of a Bernoulli distribution. We can define it more formally as follows:

The *Bernoulli distribution* is a discrete probability distribution of a random variable that takes two values, $\{0, 1\}$, with probabilities $p$ and $q = 1 - p$, respectively. We will say that our distribution is a $Bern(p)$.

If $k$ is a possible outcome, we can define the probability mass function $f$ of a Bernoulli distribution as:

$$f(k,p) = \begin{cases} p, & \text{if } k = 1, \\ 1-p, & \text{if } k = 0. \end{cases}$$

Using the expression of the mean for discrete random variables, we obtain that $E[X] = p$ and

$$\text{Var}[X] = E[X^2] - E[X]^2 = E[X] - E[X]^2 = p - p^2 = p(1-p) = pq.$$

As a note, this is just a particular case of the *Binomial distribution* with $n = 1$.

#### *Gaussian Distribution*

The Gaussian (or normal) distribution is used to represent real-valued random variables whose distributions are not known. Its importance relies in the fact that, using the *central limit theorem*, we can assume that the average of many samples of a random variable with finite mean and variance is a random variable whose distribution converges to a normal distribution as the number of samples increases.

**Definition 2.2.1.** We say that the real valued random variable $X$ follows a *normal distribution* of parameters $\mu, \sigma \in \mathbb{R}$ if, and only if, its probability density function exists and it is determined by

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}, \tag{1}$$

where $\mu$ is the mean and $\sigma$ is its standard deviation. We denote this normal distribution as $X \sim \mathcal{N}(\mu, \sigma)$.

The particular case where $\mu = 0$ and $\sigma = 1$ is widely used in statistics. In this case, the density function is simpler:

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}.$$

A remarkable property of these distributions is that, if $f : \mathbb{R} \to \mathbb{R}$ is a real-valued function defined as $f(x) = ax + b$, then $f(X) \sim \mathcal{N}(a\mu + b, |a|\,\sigma)$.

In the same way that we extended random variables to random vectors, we can extend the normal distribution to a multivariate random distribution.

**Definition 2.2.2.** We say that a random vector $\boldsymbol{X} = (X_1, \ldots, X_n)$ follows a multivariate normal distributions of parameters $\mu \in \mathbb{R}^n$, $\Sigma \in \mathcal{M}_N(\mathbb{R})$ if, and only if, its probability density function is:

$$f(x) = \frac{1}{\sqrt{\det(2\pi\Sigma)}} e^{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)}.$$

It is denoted $X \sim \mathcal{N}(\mu, \Sigma)$. In this case, $\mu$ is the mean vector of the distribution and $\Sigma$ denotes the covariance matrix.

## 2.3 STATISTICAL INFERENCE

Statistical inference is the process of deducing properties of an underlying distribution by analyzing the data that it is available. With this purpose, techniques like deriving estimates and testing hypotheses are used.

Inferential statistics are usually contrasted with descriptive statistics, which are only concerned with properties of the observed data. The difference between these two is that in inferential statistics, we assume that the data comes from a larger population that we would like to know.

In *machine learning*, subject that concerns us the most, the term inference is sometimes used to mean *make a prediction by evaluating an already trained model*, and in this context, inferring properties of the model is refered as *training or learning*.

### 2.3.1 Parametric Modeling

In the following chapters, we will be trying to estimate density functions in a dataset. To do this we will be using *parametric models*. We say that a *parametric model*, $P_\theta(x)$, is a family of density functions that can be described using a finite numbers of parameters $\theta$. We can get to the concept of *log-likelihood* now.

**Definition 2.3.1.** The *likelihood* $\mathcal{L}(\theta|x)$ of a parameter set $\theta$ is a function that measures how plausible is $\theta$, given an observed point $x$ in the dataset $\mathcal{D}$. It is defined as the value of the density function parametrized by $\theta$ at $x$. That is:

$$\mathcal{L}(\theta|x) = P_\theta(x).$$

In a finite dataset $\mathcal{D}$ consisting of independent observations, we can write:

$$\mathcal{L}(\theta|X) = \prod_{x \in D} P_\theta(x).$$

In practice, it is ofter convenient to work with the natural logarithm of the likelihood function.

**Definition 2.3.2.** Let $\mathcal{D}$ be a dataset of independent observations and $\theta$ a set of parameters. Then, we define the *log-likelihood* $\ell$ as the sum of the logarithms of the evaluations of $p_\theta$ in each $x$ in the dataset. That is:

$$\ell(\theta|X) = \sum_{x \in \mathcal{D}} \log P_\theta(x).$$

Since the logarithm is a monotonic function, the maximum of $\mathcal{L}$ and the maximum of its logarithm will occur at the same $\theta$. Our goal would be to find the optimal value $\hat{\theta}$ that maximizes the likelihood of observing the dataset $\mathcal{D}$. We get to the following definition:

**Definition 2.3.3.** We say that $\hat{\theta} = \hat{\theta}(\mathcal{D})$ is a *maximum likelihood estimator*(MLE) for $\theta$ if

$$\hat{\theta} \in \arg\max_\theta \mathcal{L}(\theta|\mathcal{D})$$

for every observation $\mathcal{D}$.

Usually, we seek for likelihood functions that are differentiable, so the derivative test for determining maxima can be applied. Sometimes, the first-order conditions of the likelihood function can be solved explicitly, like in the case of the ordinary least squares estimator which maximizes the likelihood of a linear regression model. However, most of the times, we have to make use of numerical methods to be able to find the maximum of the likelihood function.

There is another concept related to the probability and the set of parameters that the distribution takes:

**Definition 2.3.4.** The *prior probability* is the probability distribution that it is believed to exist before evidence is taken into account.

The *posterior probability* is the probability of the parameters $\theta$ given the sampled data $X$, that is, $P(\theta, X)$.

The relation with the likelihood function $P(X|\theta)$ is that, given a prior belief that a p.d.f. is $P(\theta)$ and observations $x$ have a likelihood $P(x|\theta)$, the posterior probability is defined using the prior probability as follows:

$$P(\theta|x) = \frac{P(x|\theta)}{P(x)}P(\theta),$$

where we have simply used Bayes' theorem.

### 2.3.2 Generative Models

The vast majority of the problems in ML are usually of a discriminative nature, which is almost a synonym of supervised learning. However, there also exist problems that involve learning how to generate new examples of the data. More formally:

**Definition 2.3.**
1. *Discriminative models* estimate $p(y|x)$, the probability of a label $y$ given an observation $x$.
2. *Generative models* estimate $p(x)$, the probability of observing the datapoint $x$. If the dataset is labeled, a generative model can also estimate the distribution $p(x|y)$.

From now on, let $\mathcal{D}$ be any kind of observed data. This will always be a finite subset of samples taken from a probability distribution $p_{\text{data}}$. There are models that, given $\mathcal{D}$, try to approximate the probability distribution that lies underneath it. These are called *generative models (G.M.)*.

Generative models can give parametric and non parametric approximations to the distribution $p_{\text{data}}$. In our case, we will focus on parametric approximations where the model searches for the parameters that minimize a chosen metric (which can be a distance or other kind of metric such as K-L divergence) between the model distribution and the data distribution.

We can express our problem more formally as follows. Let $\theta$ be a generative model within a model family $\mathcal{M}$. The goal of generative models is to optimize:

$$\min_{\theta \in \mathcal{M}} d(p_{\text{data}}, p_\theta),$$

where $d$ stands for the distance between the distributions. We can use, for instance, K-L divergence.

Generative models have many useful applications. We can however remark the tasks that we would like our generative model to be able to do. Those are:

- Estimate the density function: given a datapoint, $x \in D$, estimate the probability of that point $p_\theta(x)$.
- Generate new samples from the model distribution $x \sim p_\theta(x)$.
- Learn useful features of the datapoints.

If we have a look again at the example of the zebras, if we make our generative model learn about images of zebras, we will expect our $p_\theta(x)$ to be high for zebra's images. We will also expect the model to generate new images of this animal and to learn different features of the animal, such as their big size in comparison with cats.

We have to remark an example of generative models since it will be mentioned later. In time-series theory, *autoregressive models (AR)* are feed-forward models that predict future values using past values.

### 2.3.3 Minimal sufficient statistics

In parametric modeling, the goal was to determine the density function under a distribution. Another interesting task can be determining specific parameters or quantities related to a distribution, given a sample $X = (x_1, \cdots, x_n)$.

> **Definition 2.3.5.** Let $(\Omega, \mathscr{A})$ be a measurable space where $\mathscr{A}$ contains all singletons. A statistic is a measurable function of the data, that is: $T : X \to \Omega$ where $T$ is measurable.

*Remark* 1. A statistic is also a random variable.

However, not all statistics will provide useful information for the statistical inference problem, since almost anything can be a statistic. We would like to find statistics that provide relevant information.

> **Definition 2.3.6.** Let $X \sim P_\theta$. Then, the statistic $T(X) = T : (\Omega, \mathscr{A}) \to (\mathbb{T}, \mathcal{B})$, is sufficient for a family of parameters $\{P_\theta \ : \ \theta \in \Theta\}$ if the conditional distribution of $X$, given $T = t$, is indepentent of $\theta$.

*Example* 3. The simplest example of a sufficient statistic is the mean $\mu$ of a gaussian distribution with known variance. Oppositely, the *median* of an arbitrary distribution is not sufficient for the mean since, even if the median of the sample is known, more information about the mean of the population can be obtained from the mean of the sample itself.

Although it will not be shown in this document, sufficient statistics are not unique. In fact, if $T$ is sufficient, $\psi(T)$ is sufficient for any bijective mapping $\psi$. It would be interesting to find a sufficient statistic $T$ that is *the smallest* of them.

> **Definition 2.3.7.** A sufficient statistic $T$ is minimal if, for every sufficient statistic $U$, there exists a mapping $f$ such that $T(x) = f(U(x))$ for any $x \in \Omega$.

## 2.4 INTRODUCTION TO DEEP LEARNING

In order to be able to understand the chapters that will come later in this document, it is important to make a brief introduction of what *Deep Learning*(DL) refers to. Deep learning is included in the field of Machine Learning, which is also included in the field of general Artificial Intelligence.

In Chapter 1, an intuitive definition of what Machine Learning is was given. We said that ML studies the algorithms that improve from experience. Tom M. Mitchell (Mitchell, 1997) provided a more formal definition of what *learning from experience* means:

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

We would also like to have a DL definition. In Deng (2014), multiple similar definitions are given. We present here the simplest of them:

> **Definition 2.5.** *Deep Learning* is a class of ML learning techniques that exploit many layers of non-linear information processing for supervised or unsupervised feature extraction and transformation, and for pattern analysis and classification.

Usually, these techniques are based on the biologically inspired *neural networks*(NNs), which consists of several connected units: the *neurons*. Each neuron is basically a Perceptron, which is a is a weighted sum followed by a non-linear function, called an *activator* in the ML context. Formally, the output of each neuron is

$$y = \phi\left(w_0 + \sum_{i=1}^{N} w_i x_i\right).$$

There are many activation functions, but the following examples must be remarked:

- Sigmoid. The sigmoid function is defined as follows:

$$\phi(x) = \frac{1}{1+e^{-x}}.$$

  This is one of the most common used activation functions. It is differentiable, monotonic and smooth. One of its main disadvantages is that at the right part of the function, the change in the values that the

function takes converges to zero, so we get to the *vanishing gradient* problem and the learning is minimal.

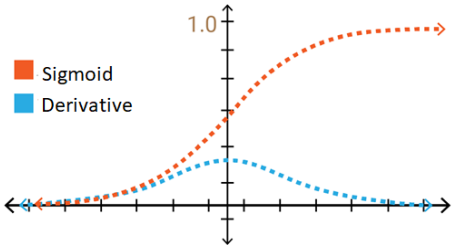



**Figure 5:** Sigmoid. Image from this Medium article.

- Hyperbolic Tangent. This function is defined as follows:

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

This activation function has a small advantage over the sigmoid: its derivative is more steep, which means it can get more value and the learning can be more efficient.

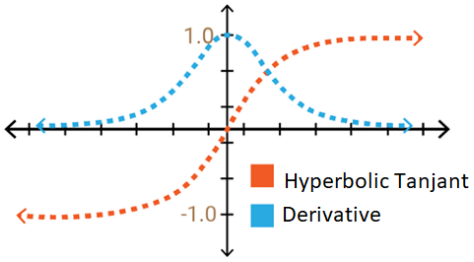



**Figure 6:** Hyperbolic Tangent. Image from this Medium article.

- Rectified Linear Unit (ReLU). This function takes the following form:

$$\phi(x) = \max(0, x).$$

ReLu is highly computationally efficient and non-linear. Its main problem is that when the inputs approach zero or are negative, the network can not perform back propagation and can not learn.

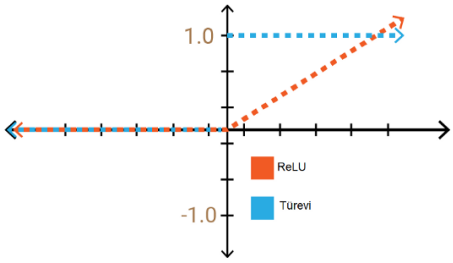



**Figure 7:** ReLU. Image from this Medium article.

### 2.4.1 Neural Networks

Using neurons and activation functions, we can formally define NNs. A NN with $L$ hidden layers is a deterministic non-linear function $f$, parametrized

by a set of matrices $W = \{W_0, \cdots, W_L\}$ and non-linear activation functions $\{\phi_0, \cdots, \phi_L\}$. Given an input $x$, the output $y$ of the network is calculated as follows:

$$h_0 = \phi_0\left(W_0^T x\right), \cdots, h_l = \phi_l\left(W_l^T h_{l-1}\right), \cdots, y = \phi_L\left(W_L^T h_{L-1}\right).$$

Having a NN, we consider it *deep* when the number of hidden layers (and, consequently, the number of matrices) is considered high.

Neural networks use loss functions, which define how well the output returned by the network matches the real output, reducing the learning problem to an optimization problem. The problem is finding $W^{\text{opt}}$, such that

$$W^{\text{opt}} = \arg\min_w \sum_{n=1}^{N} l(y_n, f_w(x_n)),$$

where $\mathcal{D} = \{(x_n, y_n)\}$ is a dataset.

This problem is solved using a variant of *stochastic gradient descent (SGD)*. This algorithm involves the computation of the loss function $l$ derivatives respect to the network parameters, and updates the parameters using this derivatives. Specifically, the parameters are updated as follows:

$$W_{t+1} = W_t + \eta \nabla l(W_t),$$

where $\eta \in \mathbb{R}^+$ is a small constant called the *learning rate*. This algorithm guarantees convergence to local minimums of $f$ and, if $f$ is convex, the algorithm converges to a global minimum.

The last comment about neural networks is that, since the weights $W = \{W_0, \cdots, W_L\}$ are constantly updated, the derivatives have to be computed repeatedly. The computational cost of this is quite high. *Backpropagation* was born to calculate the derivatives of the weights much faster. The intuitive idea is that the gradient of the layer $l$ is computed using the gradient of the layer $l + 1$ using the chain rule.

Understanding both SGD and Backpropagation is crucial for understanding how NNs work. However, in the experimentation part of this work we will focus on researching how a few hyperparameters affect the results of the proposed frameworks, so no further explanation on these important concepts will be provided.

### 2.4.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a specific type of Neural Networks. The difference that we have between CNNs and NNs is that CNNs assume that the inputs have local dependencies. For instance, using an image, a CNN assumes (most of the time correctly) that given a pixel $x_{ij}$ of an input $x$, the neighbours of this pixels will have similar values or intensities. This allows us to encode certain properties into the architecture of the network.

If we use colored images, the input of the CNNs are 3-dimensional volumes, which will be transformed in some layers to other 3-dimensional volumes.

In order to do this we have different types of layers, remarking *convolutional* layers, *pooling* layers and *fully connected* layers. The most important ones are convolutional layers.

Convolutional layers receive a *tensor* with shape $k \times n \times m \times c$, which means that the layer receives $k \in \mathbb{N}$ inputs of sizes $n \times m \times c$. After the convolutional layer, the input has shape $k \times n' \times m' \times c'$, where $n'$ can be different from $n$ (respectively $m', c'$). One convolutional layer can apply one or several filters to the same input, producing different 2-dimensional activation maps that are stacked along the depth dimension to produce the output.

Formally, a convolution is the process of adding each element of the image to its local neighbours, weighted by a kernel. That is in our case performing a dot product between the input and the filter. We obtain

$$g(x,y) = \omega \star f(x,y) = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} \omega(dx,dy) f(x+dx, y+dy),$$

where $g(x,y)$ is the pixel $(x,y)$ of the filtered image, $f(x,y)$ is the same pixel in the original image, and $\omega$ is the filter kernel. Depending on the filter kernel, the result will be different.

*Example* 4. If we want to produce noise reduction in an image, we use Gaussian Blur, which kernel is calculated by a Gaussian function like the one in Equation 1. For instance, a $3 \times 3$ gaussian kernel would approximately be:

$$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}.$$

### 2.4.3 ResNet

To finish with this introduction to CNNs, we will present a widely used CNN architecture. Since CNNs appeared, people tried to build such deep networks that were very hard to train. When deeper networks start converging, a degradation problem ocurred: with the network depth increasing, accuracy gets saturated and then it degraded rapidly. *ResNet* (He *et al.*, 2015) brought and end to this problem, allowing the training of very deep models with up to hundreds of layers.

In the paper, they mentioned that if $\mathcal{H}(x)$ is the underlying map of a sequence of layers and knowing that it can be asymptotically approximated, we can also approximate its residuals

$$\mathcal{F}(x) = \mathcal{H}(x) - x.$$

Then, the original function becomes $\mathcal{F}(x) + x$. In order to achieve this, they introduced the *Residual Block*, a block of layers that introduces skip or shortcut connections that makes it easy for networks to represent the identity mapping. We can see one of this blocks depicted in Figure 8.

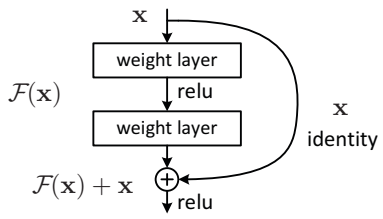



**Figure 8:** Residual Block. Image from He *et al.* (2015)

Usually, ResNet comes accompanied by a number (e.g. *Resnet-50*), that indicates the number of layers it has. Usually, a *batch normalization* is used after each convolution and ReLU is used at the end of each group of layers.

| Layers | Output Size | Resnet 18 |
|---|---|---|
| conv1 | $112 \times 112$ | $7 \times 7,\ 64$, stride 2 |
| conv2 | $56 \times 56$ | $3 \times 3$ max pool, stride 2 |
| | | $\begin{bmatrix} 3 \times 3, & 64 \\ 3 \times 3, & 64 \end{bmatrix} \times 2$ |
| conv3 | $28 \times 28$ | $\begin{bmatrix} 3 \times 3, & 128 \\ 3 \times 3, & 128 \end{bmatrix} \times 2$ |
| conv4 | $14 \times 14$ | $\begin{bmatrix} 3 \times 3, & 256 \\ 3 \times 3, & 256 \end{bmatrix} \times 2$ |
| conv5 | $7 \times 7$ | $\begin{bmatrix} 3 \times 3, & 512 \\ 3 \times 3, & 512 \end{bmatrix} \times 2$ |
| | $1 \times 1$ | average pool, 1000-d fc, softmax |
| Number of parameters | | 11.400.000 |

**Table 1:** Resnet 18 architecture.

For instance, having a look at the Table 1, in the convolution block *conv2*, we have two convolutions with kernel size $3 \times 3$ and depth size 64. This would be one of the blocks, represented in Figure 8. However, since at the right hand side of the matrix we have a $\times 2$, it means that we will have two of this blocks in this convolution group.

### 2.4.4 Data augmentation

In the general problem of performing a DL task, it may happen that the amount of data, either labeled or unlabeled, is not enough to give the model the number of examples that it needs to learn.

> **Definition 2.4.1.** *Data augmentation* are techniques used to increase the amount of data by adding modified copies of the already existing data or newly created synthetic data also from the existing one.

In our self-supervised problem, data augmentation gains even more importance. It has been empirically shown that chosing the appropiate techniques

and the order of the aplication of them to the images can cause a huge improvement on the results (Chen *et al.*, 2020b).

Depending on the kind of data that we are facing, the techniques change. For instance, if we are working in the field of Natural Language Processing (NLP), which involves applying ML to texts, we should use techniques such as *Back translation* or *Synonym replacement*. If we try to process audio we can use *Noise Injection* or changing the *Speed* of the audio.

In our case, we will be working with images. Let us present some examples that can be used for data augmentation in the computer vision field.

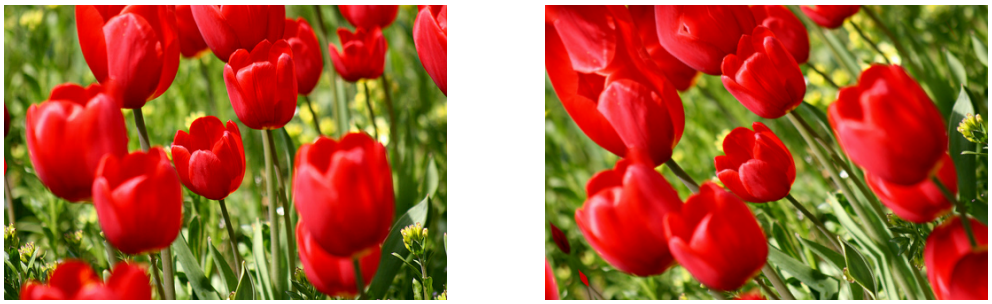
#### *Rotations and flips*

A very common data augmentation technique is the *rotation* of the image a certain amount of degrees, usually one of the following: $\left\{\frac{\pi}{2}, \pi, \frac{3}{2}\pi\right\}$, using the center of the image as a rotation center. To do this, we only have to consider the following transformation:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix},$$

where $(x, y)$ is the pixel of the image that we will rotate and $\theta$ are the degrees that the image will be rotated.

This rotation can be composed (or not) with a *flip* operation. That operation consists of applying a symmetry respect the central column of an image. Clearly, a flip can be applied on its own and it is still generating a new example.



**(a)** Tulips **(b)** Rotated, flipped tulips

**Figure 9:** The image on the left is the original image, and the image on the right is a new example generated by first a rotation of a random angle and then a flip.

#### *Random crop with resize*

Another type of example generation is taking a crop randomly out of the image, making sure that the whole crop stays inside the image, and then resizing it back to the original image size.

If an image is sized $n \times m$ with $n, m \in \mathbb{N}$, this method consists of selecting a rectangle of size $n' \times m'$ with $n' \leq n, m' \leq m$ of the original image and then resizing it back via upsampling[1].

**(a)** Tulips

**(b)** Random cropped, resized tulips

**Figure 10:** The image on the left is the original image, and the image on the right is a new example generated by performing a random crop of the image and then resizing the crop to the original image size.

This can be useful to obtain images that are similar to each other, for instance, taking two crops of the same image. However, this could also lead to the model to learn an specific kind of data that is not relevant in general: the image data histogram. Let us see the image data histograms for both the original image and the random cropped image:

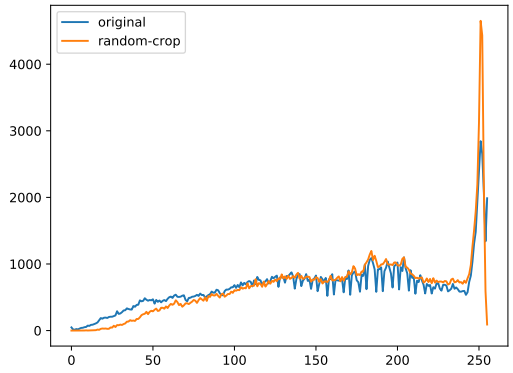



**Figure 11:** Histogram for both the original image on blue and the random crop on orange.

As we can see, the histograms of both the images are quite similar, so a DL model could learn to indentify similar objects making use only on this data histogram, which would not be very effective since the same object can have many different histograms. Because of this, this random crops are useful but the random cropped images have to be applied more functions in order to be really useful for the training of our models.

1 Upsampling is increasing the size of an image by inserting new rows and columns in the image matrix and interpolating the values of the introduced pixels using the pixels that we already had in the image.

### *Random color jitter*

This kind of data augmentation is not always used, but it is very interesting in our case. The random crops produced very similar data histograms since the colors in the original image and the cropped image are very similar. We can try to change this by randomly modifying the value of the pixels by changing the brightness, saturation or the contrast of the image.

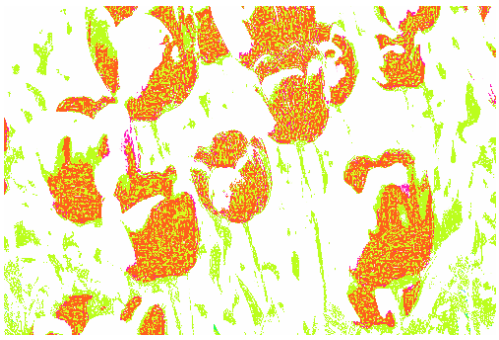


**Figure 12:** Color jitter applied to tulips image.

We would like to see if there is a difference between the histograms of the cropped image compared with the color jittered image. The result is the following:

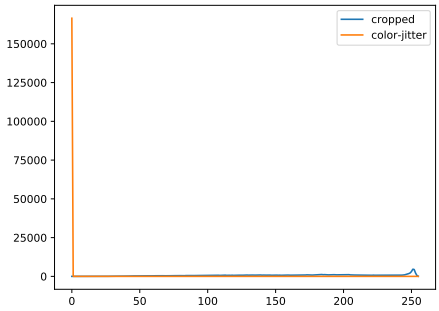



**Figure 13:** Histograms of the color jitter image in orange and cropped image in blue.

As we can see, since in the color jittered image most of the pixels turn white, the histograms get a big difference this time specially in the left side. This way, if we use both augmentation composed, we will avoid our model to learn only from the data histogram of our images.

# 3 INFORMATION THEORY FUNDAMENTALS

## 3.1 ENTROPY

Obtaining good representations of data is one of the most important tasks in machine learning. Recently, it has been discovered (Oord *et al.*, 2019) that maximizing the *mutual information* between two elements in our data can give us good representations for our data. In this section, *information theory* notions will be presented, in order to use them in our ML models. This will provide a theoretical solid base for the notions explained later.

The *mutual information* concept is based on the *Shannon entropy*, which we will introduce first, along with some basic properties of it. The Shannon entropy is a way of measuring the uncertainty in a random variable. Given an event $\mathcal{A} \in \mathscr{A}$, $P$ a probability measure and $P[\mathcal{A}]$ the probability of $\mathcal{A}$, we can affirm that

$$\log \frac{1}{P[\mathcal{A}]}$$

describes *"how surprising is that $\mathcal{A}$ occurs"*. Clearly, then,

- If $P[\mathcal{A}] = 1$, then we have $\log 1 = 0$, which would indicate that it is not a surprise that $\mathcal{A}$ occurred.
- On the other hand, if $P[A] \to 0$, then we have $\log(+\infty)$, which will surely be a big number indicating great surprise that $\mathcal{A}$ occurred.

With this motivation, we get to the following definition.

**Definition 3.1.1.** Let $X$ be a discrete random variable with image $\mathcal{X}$. The *Shannon entropy*, or simply *entropy* $H(X)$ of $X$ is defined as:

$$H(X) = E_X\left[\log \frac{1}{P_X(X)}\right] = \sum_{x \in \mathcal{X}} P_X(x) \log \frac{1}{P_X(x)}.$$

The *entropy* can trivially be expressed as:

$$H(X) = -\sum_{x \in \mathcal{X}} P_X(x) \log P_X(x).$$

This simple example, (Cover & Thomas, 1991), even though it is very simple, it is very illustrative for our definition:

*Example* 5. Let $X \sim Bern(p)$. Then, the entropy of $X$ is:

$$H(X) = -p \log p - (1-p) \log(1-p)) = H(p),$$

since $H$ only depends on $p$. In Fig. 5 we can see a representation of this function. We appreciate that in this case, $H$ is concave and equals 0 if $p \in$

$\{0,1\}$, which are the values of $p$ that give us no uncertainty. The maximum uncertainty is obtained when $p = \frac{1}{2}$, where we do not know what to expect as an outcome from our random variable $X$.

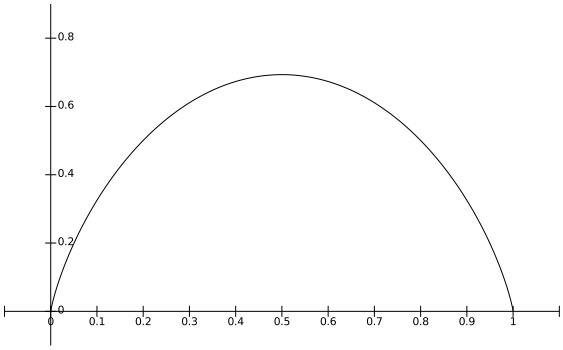



**Figure 14:** Representation of $H(p)$ in the example 5.

It can also be proven that, in general, the entropy is concave.

### 3.1.1 Properties of the entropy. Conditional entropy

There are some properties of the *entropy* that must be remarked, since they will extend to properties of the mutual information.

> **Proposition 2.** *Let $X$ be a random variable with image $\mathcal{X}$. If $|\mathcal{X}|$ is the cardinal of $\mathcal{X}$, then*
>
> $$0 \leq H(X) \leq \log(|\mathcal{X}|).$$

*Proof.* Since $\log y$ is concave on $\mathbb{R}^+$, by Jensen's inequality ( see Appendix A, Prop. 9), we obtain:

$$H(X) = -\sum_{x\in\mathcal{X}} P_X(x)\log P_X(x) \leq \log\left(\sum_{x\in\mathcal{X}} 1\right) = \log(|\mathcal{X}|).$$

For the lower bound we see that, since $P_X(x) \in [0,1]$ for all $x \in \mathcal{X}$ then $\log P_X(x) \leq 0 \quad \forall x \in \mathcal{X}$. Hence , $-P_X(x)\log P_X(x) \geq 0$ for all $x \in X$, so $H(X) \geq 0$. □

We can also see that the equality on the left holds if , and only if , exists $x$ in $X$ such that its probability is exactly one, that is $P_X(x) = 1$. The right equality holds if and only if , for all $x \in \mathcal{X}$, its probability is $P_X(x) = \frac{1}{|X|}$.

#### Conditional entropy

We have already said that entropy measures how surprising is that an event occurs. Usually, we will be looking at two random variables and it would be interesting to see how likely is that one of them, say $X(x)$, occurred, if we already know that $Y(y)$ occurred. This leads us to the definition of *conditional entropy*. Let us see a simpler case first:

Let $A$ be an event, and $X$ a random variable. The conditional probability $P_{X|A}$ defines the entropy of $X$ conditioned to $A$:

$$H(X|A) = \sum_{x \in \mathcal{X}} P_{X|A}(x) \log \frac{1}{P_{X|A}(x)}.$$

If $Y$ is another random variable and $\mathcal{Y}$ is its image, intuitively we can sum the conditional entropy of an event with all the events in $\mathcal{Y}$, and this way we obtain the conditional entropy of $X$ given $Y$.

**Definition 3.1.2** (Conditional Entropy)**.** Let $X, Y$ be random variables with images $\mathcal{X}, \mathcal{Y}$. The *conditional entropy* $H(X|Y)$ is defined as:

$$\begin{aligned} H(X|Y) &:= \sum_{y \in \mathcal{Y}} P_{\mathcal{Y}}(y) H(X|Y = y) \\ &= \sum_{y \in \mathcal{Y}} P_{\mathcal{Y}}(y) \sum_{x \in \mathcal{X}} P_{X|Y}(x|y) \log \frac{1}{P_{X|Y}(x|y)} \\ &= \sum_{x \in X, y \in \mathcal{Y}} P_{XY}(x,y) \log \frac{P_Y(y)}{P_{XY}(x,y)}. \end{aligned}$$

The interpretation of the conditional entropy is simple: the uncertainty in $X$ when $Y$ is given. Since we know about an event that has occurred ($Y$), intuitively the conditional entropy , or the uncertainty of $X$ occurring given that $Y$ has occurred, will be lesser than the entropy of $X$, since we already have some information about what is happening. We can prove this:

**Proposition 3.** *Let $X, Y$ be random variables with images $\mathcal{X}, \mathcal{Y}$. Then:*

$$0 \leq H(X|Y) \leq H(X).$$

*Proof.* The inequality on the left was proved on Proposition 2. The characterization of when $H(X|Y) = 0$ was also mentioned after it. Let us look at the inequality on the right. Note that restricting to the $(x, y)$ where $P_{XY}(x, y) > 0$ and using the definition of the conditional probability we have:

$$\begin{aligned} H(X|Y) &= \sum_{y \in \mathcal{Y}} P_Y(y) \sum_{x \in \mathcal{X}} P_{X|Y}(x|y) \log \frac{1}{P_{X|Y}(x|y)} \\ &= \sum_{x \in \mathcal{X}, y \in \mathcal{Y}} P_Y(y) P_{X|Y}(x,y) \log \frac{P_Y(y)}{P_{XY}(x,y)} \\ &= \sum_{x \in \mathcal{X}, y \in \mathcal{Y}} P_{XY}(x,y) \log \frac{P_Y(y)}{P_{XY}(x,y)}, \end{aligned}$$

and

$$H(X) = \sum_x P_X(x) \log \frac{1}{P_X(x)} = \sum_{x,y} P_{XY}(x,y) \log \frac{1}{P_X(x)}.$$

Hence,

$$\begin{aligned} H(X|Y) - H(X) &= \sum_{x,y} P_{XY}(x,y) \left( \log \frac{P_Y(y)}{P_{XY}(x,y)} - \log \frac{1}{P_X(x)} \right) \\ &= \sum_{x,y} P_{XY} \log \frac{P_Y(y) P_X(x)}{P_{XY}(x,y)}. \end{aligned} \tag{2}$$

So, using Jensen's inequality, we obtain:

$$\sum_{x,y} P_{XY} \log \frac{P_Y(y)P_X(x)}{P_{XY}(x,y)} \leq \log\left(\sum_{x,y} \frac{\cancel{P_{XY}(x,y)}\ P_Y(y)P_X(x)}{\cancel{P_{XY}(x,y)}}\right)$$
$$= \log\left(\left(\sum_x P_X(x)\right)\left(\sum_y P_Y(y)\right)\right) = \log 1 = 0,$$

and this leads us to:

$$H(X|Y) - H(X) \leq 0 \quad \text{then} \quad H(X|Y) \leq H(X) \tag{3}$$

as we wanted. □

It must be noted that the inequality the state of the proposition,

$$0 \leq H(X|Y) \leq H(X),$$

in the inequality of the left, equality holds if, and only if, $P_{XY}(x,y) = P_X(x)P_Y(y)$ for all $(x,y)$ with $P_{XY}(x,y) > 0$, as it is said in Jensen's inequality. For the inequality on the right, equality holds if and only if $P_{XY}(x,y) = 0$, which implies $P_X(x)P_Y(y) = 0$ for any $x \in \mathcal{X}$, $y \in \mathcal{Y}$. It follows that $H(X|Y) = H(X)$ if and only if $P_{XY}(x,y) = P_X(x)P_Y(y)$ for all $(x,y) \in \mathcal{X} \times \mathcal{Y}$

## 3.2 MUTUAL INFORMATION

Using the entropy of a random variable we can directly state the definition of *mutual information* as follows:

> **Definition 3.1.** Let $X, Z$ be random variables. The *mutual information (MI)* between $X$ and $Z$ is expressed as the difference between the entropy of $X$ and the conditional entropy of $X$ and $Z$, that is:
>
> $$I(X,Z) := H(X) - H(X|Z).$$

Since the entropy of the random variable $H(X)$ explains the uncertainty of $X$ occurring, the intuitive idea of the *MI* is to determine the decrease of uncertainty of $X$ occurring when we already know that $Z$ has occurred. We also have to note that, using the definition of the *entropy* and the same argument that we used to obtain the expression in Eq. 2, we can rewrite the *MI* it follows:

$$I(X,Z) = \sum_{x\in\mathcal{X}} P_X(x) \log \frac{1}{P(x)} - \sum_{x\in\mathcal{X}, z\in\mathcal{Z}} P_{XZ}(x,z) \log \frac{P_Z(x)}{P_{XZ}(x,z)}$$
$$= \sum_{x,z} P_{XZ} \log \frac{P_{XZ}(x,z)}{P_Z(z)P_X(x)} \tag{4}$$

and if we compare it to the formula of the KL-Divergence, we obtain:

$$I(X,Z) = \sum_{x,z} P_{XZ} \log \frac{P_{XZ}(x,z)}{P_Z(z)P_X(x)} = D_{KL}(P_{XZ} \,||\, P_X P_Z),$$

so we have obtained an expression of the mutual information using the *Kullback-Leibler* divergence. This provides with the following immediate consequences:

**Corollary 1.** *(i) Mutual information is non-negative. That is :* $I(X,Z) \geq 0$.
*(ii) If* $X, Z$ *are random variables, then its mutual information equals zero if, and only if, they are independent.*
*(iii) Mutual information is symmetric. That is:* $I(X,Z) = I(Z,X)$.

*Proof.* *(i)* This is trivial using Prop 3 and the definition of the mutual information.
*(ii)* We can use the KL-Divergence formulation to see that since

$$D_{KL}(P_{XZ} \,||\, P_X P_Z) = 0 \implies P_{XZ} = P_X P_Z,$$

almost everywhere then $X$ and $Z$ are independent.
*(iii)* It is a consequence of the fact that $P_{XZ} = P_{ZX}$ and $P_X P_Z = P_Z P_X$.

□

Later in this document, we will have some sort of random variable $X$ and would like it to maintain the mutual information with itself after being applied a function. The following proposition will be useful:

**Proposition 4.** *Let* $X, Z$ *be random variables. Then,* $I(X,Z)$ *is invariant under homeomorphism.*

*Proof.* Let $\phi(x)$ be an homeomorphism, i.e., a continuous, monotonic function with $\phi^{-1}(x)$ also continuous and monotonic. Let $X$ be a random variable and $Y$ another one such $y = \phi(x)$ if $x = X(\omega)$ for some $\omega \in \Omega$. Then, if $S$ is a particular subset we have

$$P(Y \in S) = \int_S P_Y(y)dy = \int_{\phi^{-1}(S)} P_X(x)dx \overset{(1)}{=} \int_S P_X(\phi^{-1}(y)) \left| \frac{d\phi^{-1}}{dy} \right| dy,$$

where in $(1)$ we have changed from $x$ to $y$. Hence,

$$P_Y(y) = P_X(\phi^{-1}(y)) \left| \frac{d\phi^{-1}}{dy} \right|.$$

As a consequence of this, $I(X,Z) = I(\phi(X),Z)$ for any homeomorphism $\phi$. By symmetry, the same holds for $Z$.

□

*Remark* 2. We can set a connection between the mutual information and sufficient statistics. Let $T(X)$ be a statistic. We say that $T(X)$ is sufficient for $\theta$ if its mutual information with $\theta$ equals the mutual information between $X$ and $\theta$, that is:

$$I(\theta, X) = I(\theta, T(X)).$$

This means that sufficient statistics preserve mutual information and conversely.

### 3.2.1 Lower bounds on Mutual Information

Although mutual information seems like a relatively intuitive concept, it is most of the times extremely hard to compute it in real life problems in which the distributions $P(x,z), P(x), P(z)$ are not known.

*Example* 6. Let $x$ represent an image of size $n \times m$ pixels. Then, the dimension of the single image is $n \cdot m \cdot 3$, for RGB color channels. In these cases, there is no easy way of calculating $P(x)$.

Due to this problem related to the *Curse of Dimensionality*, we can try to compute lower bounds of it that are generally easier to calculate. We will now expose two general lower bounds, and we will focus on a third one that will be explained later in this work.

#### *3.2.1.1 Variational Lower Bound*

Using the expression of the mutual information in terms of entropy, $I(x,z) = H(z) - H(z|x)$, we can give a lower bound on $I(x,z)$ as a function of a probability distribution $Q_\theta(z|x)$.

**Proposition 5.** *Let $X, Z$ be random variables and $Q_\theta(z|x)$ be an arbitrary probability distribution. Then,*

$$I(x,z) \geq H(z) + E_{P_X}\left[E_{P_{X|Z}}\left[\log Q_\theta(z|x)\right]\right]$$

*Proof.* Recalling that

$$H(z|x) = -E_{P_{XZ}}\left[\log P(x,z) - \log P(x)\right],$$

and that

$$\begin{aligned} E_{P(x,z)}\left[\log \frac{P(x,z)}{P(x)}\right] &= \sum_{x,z} P(x,z) \log \frac{P(x,z)}{P(x)} \\ &= \sum_{x,z} P(x)P(z|x)\log P(z|x) = \sum_{x,z} P(x) E_{P(z|x)}[\log P(z|x)] \\ &= E_{P(x)}\left[E_{P(z|x)}[\log P(z|x)]\right], \end{aligned}$$

we only have to use the definition of the conditional probability to see that:

$$\begin{aligned} I(x,z) &= H(z) - H(z|x) \\ &= H(z) + E_{P(x,z)} = H(z) + E_{P(x,z)}\left[\log \frac{P(x,z)}{P(x)}\right] \\ &= H(z) + E_{P(x)}\left[E_{P(x|z)}\left[\log P(z|x)\right]\right] \\ &= H(z) + E_{P(x)}\left[E_{P(x|z)}\left[\log \frac{P(z|x)}{Q_\theta(z|x)}\right] + E_{P(z|x)}\left[\log Q_\theta(z|x)\right]\right] \\ &= H(z) + E_{P(x)}\left[\underbrace{D_{KL}(P(z|x)||Q_\theta(z|x))}_{\geq 0} + E_{P(z|x)}\left[\log Q_\theta(z|x)\right]\right] \\ &\geq H(z) + E_{P(x)}\left[E_{P(z|x)}\left[\log Q_\theta(z|x)\right]\right]. \end{aligned}$$

We have taken advantage of the non-negativity of the KL-Divergence. □

Using this bound, and combining this theoretical knowledge with machine learning methods, such as *backpropagation*, we can make $Q_\theta$ be a neural network and maximize this lower bound.

#### 3.2.1.2 Donsker-Varadhan Representation lower bound

We can also give a lower bound on the mutual information using its KL-Divergence formulation. Firstly, we have to

**Theorem 3.2.1** (Donsker-Varadhan)**.** *The KL divergence admits the following dual representation:*

$$D_{KL}(P||Q) = \sup_T E_P[T] - \log E_Q\left[e^T\right],$$

*where the supremum is taken over all functions $T : \Omega \to \mathbb{R}$ such that both expectations exist.*

*Proof.* Let $T$ be a given function. We can consider the *Gibbs* distribution, which is defined by

$$G(x) = \frac{1}{Z} e^T Q(x),$$

where $Z$ is a normalization term defined by $Z = E_{Q(x)}[e^T]$. We observe by the definition of $G$ that, taking logarithms and then taking expectations respect to $P(x)$, we obtain

$$E_P\left[\log \frac{G(x)}{Q(x)}\right] = E_P[T] - \log Z = E_P[T] - \log\left(E_Q\left[e^T\right]\right) \tag{5}$$

Let $\Delta$ be the gap between $D_{KL}$ between the distributions $P$ and $Q$, and the RHS of the Equation (5),

$$\Delta := D_{KL}(P||Q) - \left(E_P[T] - \log(E_Q\left[e^T\right])\right).$$

Applying the definition of KL-Divergence and Equation (5), we obtain

$$\begin{aligned}
\Delta &= E_P\left[\log \frac{P}{Q}\right] - E_P\left[\log \frac{G(x)}{Q(x)}\right] \\
&= E_P\left[\log \frac{P}{Q} - \log \frac{G}{Q}\right] \\
&= E_P\left[\log \frac{P}{G}\right] \\
&= D_{KL}(P||G). \qquad (6)
\end{aligned}$$

Hence, since $D_{KL}(P||G) \geq 0$ for any distributions $P, G$, we have that $\Delta \geq 0$ and therefore

$$D_{KL}(P||Q) \geq E_P[T] - \log\left(E_Q\left[e^T\right]\right).$$

This inequality is preserved upon taking the supremum over the right hand side. Also, Equation (6) shows that the bound is *tight* whenever $P(x) = Q(x)$ for all $x$, namely for optimal functions $T^*$ that have the form

$$T^* = \log \frac{P(x)}{Q(x)} + C,$$

for some $C \in \mathbb{R}$.

□

Using this representation, we reach this lower bound as a corollary.

> **Corollary 2.** *Let $\mathcal{F}$ be any class of functions $T : \Omega \to \mathbb{R}$ satisfying the integrability constraints of the theorem. Then,*
>
> $$I(P, Q) = D_{KL}(P||Q) \geq \sup_{T \in \mathcal{F}} E_P[T] - \log E_Q \left[ e^T \right].$$

This bound may seem a bit odd in the context of Machine Learning. However, it is very useful. It was used in Belghazi *et al.* (2018) to estimate the mutual information between two random variables using neural networks.

Furthermore, since it is a lower bound, it can be used as a *loss function* to learn representations. In Hjelm *et al.* (2019), $T_\theta$ parametrized a neural network that acted as a discriminator between samples drawn from the joint distribution $P(x, z)$ and samples from the marginal distribution $P(x)P(z)$. They discovered that the inputs to the neural network, which were representations of the original data, must have high mutual information with the original data in order to obtain good results.

#### 3.2.1.3 Contrastive Lower Bound

In chapter 4 we presented Noise Contrastive Estimation, that tried to discriminate between elements of two different sets. One was composed of data, $X$, and the other one was composed of noise, $Y$.

Let $(x, z)$ be a data representation drawn from a distribution $P(x, z)$ and $x'$ be some other data drawn from the distribution $P(x)$. Using NCE, we should be able to say that $(x, z)$ was drawn from the distribution $P(x, z)$ (which was $P_d$ in the NCE theory) while $(x', z)$ was drawn from the product of the marginal distributions $P(x)P(z)$ (which was $P_n$ in the explanation of NCE). Let $h_\theta$ be a model that helps us to do this discrimination, with parameters $\theta$.

As we did before, we want to estimate the ratio $P_d/P_n$ of the different distributions, in this case the ratio would be $P(x, z)/P(x)P(z)$. Let $(x^*, z)$ be a pair drawn from $P(x, z)$ and $X = \{x^*, x_1, \cdots, x_{N-1}\}$, where the rest of the $N - 1$ points form pairs $(x_j, z)$ drawn from $P(x)P(z)$ the product of the marginal distribution. We can rewrite the loss 13 in a simpler expression:

$$l(\theta) = -E_X \left[ \log \frac{h_\theta(x^*, z)}{\sum_{x \in X} h_\theta(x, z)} \right]. \tag{7}$$

If we maximize this objective, $h_\theta$ learns to discriminate $(x^*, z)$ from $(x_j, z)$ for $1 \leq j < N$ and, thus, we are learning to estimate the ratio $P(x, z)/P(x)P(z)$. Let us see how maximizing $\ell(\theta)$ we are maximizing a lower bound for $I(x, z)$.

> **Proposition 6.** *Let $X = \{x^*, x_1, \cdots, x_{N-1}\}$, where $x^* \sim P(x, z)$ and the rest of them were sampled from $P(x)P(z)$. Then,*
>
> $$I(x, z) \geq -\ell(\theta) + \log N \tag{8}$$

*Proof.* Firstly, using Bayes' rule, $P(x^*, z) = P(x^*|z)P(z)$. Hence, since $h_\theta$ estimates $P(x^*, z)/P(x)P(z)$, it also estimates

$$\frac{P(x^*, z)}{P(x)P(z)} = \frac{P(x^*|z)P(z)}{P(x)P(z)} = \frac{P(x^*|z)}{P(x)}.$$

Using the definition of the log-likelihood that we see in (7), forgetting the sign for the moment, we see that

$$\begin{aligned} E_X\left[\log \frac{h_\theta(x^*, z)}{\sum_{x\in X} h_\theta(x, z)}\right] &= E_X\left[\log \frac{h_\theta(x^*, z)}{h_\theta(x^*, z) + \sum_{j=1}^{N-1} h_\theta(x_j, z)}\right] \\ &\approx E_X\left[\log \frac{\frac{P(x^*|z)}{P(x)}}{\frac{P(x^*|z)}{P(x)} + \sum_{j=1}^{N-1} \frac{P(x_j|z)}{P(x)}}\right]. \end{aligned}$$

Now, using that $\log(a) = -log(a^{-1})$,

$$\begin{aligned} E_X\left[\log \frac{\frac{P(x^*|z)}{P(x)}}{\frac{P(x^*|z)}{P(x)} + \sum_{j=1}^{N-1} \frac{P(x_j|z)}{P(x)}}\right] &= E_X\left[-\log\left(\frac{\frac{P(x^*|z)}{P(x)} + \sum_{j=1}^{N-1} \frac{P(x_j|z)}{P(x)}}{\frac{P(x^*|z)}{P(x)}}\right)\right] \\ &= E_X\left[-\log\left(1 + \frac{\sum_{j=1}^{N-1} \frac{P(x_j|z)}{P(x)}}{\frac{P(x^*|z)}{P(x)}}\right)\right] \\ &= E_X\left[-\log\left(1 + \frac{(N-1)E_{X-\{x^*\}}\left[\frac{P(x|z)}{P(x)}\right]}{\frac{P(x^*|z)}{P(x)}}\right)\right] \end{aligned}$$

Now, since $E_{X-\{x^*\}}\left[\frac{P(x|z)}{P(x)}\right] = \sum_{x_j\in X-\{x^*\}} P(x_j)\frac{P(x_j|z)}{P(x_j)} = 1$, then

$$E_X\left[-\log\left(1 + \frac{(N-1)E_{X-\{x^*\}}\left[\frac{P(x|z)}{P(x)}\right]}{\frac{P(x^*|z)}{P(x)}}\right)\right] = E_X\left[-\log\left(1 + \frac{(N-1)}{\frac{P(x^*|z)}{P(x)}}\right)\right].$$

Lastly, using that if $k > 0$, then $-\log a(k+1) \geq -\log(1+ak)$, we obtain:

$$\begin{aligned} E_X\left[-\log\left(1 + \frac{(N-1)}{\frac{P(x^*|z)}{P(x)}}\right)\right] &= E_X\left[\log \frac{1}{1 + \frac{P(x^*)}{P(x^*|z)}(N-1)}\right] \\ &\leq E_X\left[\log\left(\frac{1}{\frac{P(x^*)}{P(x^*|z)}}\frac{1}{N}\right)\right] \\ &= E_X\left[\log\left(\frac{P(x^*|z)}{P(x^*)}\frac{1}{N}\right)\right] \\ &= E_X\left[\log\left(\frac{P(x^*|z)}{P(x^*)}\right)\right] - \log N \\ &\overset{(1)}{=} E_X\left[\log\left(\frac{P(x^*, z)}{P(x^*)P(z)}\right)\right] - \log N \\ &\overset{(2)}{=} I(x, z) - \log N, \end{aligned}$$

where, in (1), we have use Bayes' rule again and in (2) we have used the definition of the MI that we found in equation 4. Looking at the first and last equations used in this proof, and seeing that we have $\leq$ in the middle of the chain of equalities, we have proved

$$E_X\left[\log \frac{h_\theta(x^*, z)}{\sum_{x \in X} h_\theta(x, z)}\right] = -\ell(\theta) \leq I(x, z) - \log N,$$

which implies

$$I(x, z) \geq -\ell(\theta) + \log N$$

as we wanted to see. □

# Part II

# CONTRASTIVE LEARNING

# 4 NOISE CONTRASTIVE ESTIMATION

In this chapter, we will present the mathematical fundamentals that inspired the methods that nowadays obtain the best results in the field of unsupervised representation learning, which is a very important part of the new advances in machine learning. All these concepts will be clarified later in this document.

## 4.1 LOGISTIC REGRESSION

The logistic model is used to model the probability of a certain *discriminative* class o event existing, such as pass/fail or dead/alive. It can be extended to a multiple class problem, but we will focus on the binary problem.

> **Definition 4.1.1.** The *odds* is a likelihood measure of a particular outcome. It is calculated as the ratio of the number of events that produce that outcome to the numbers of events that produce a different outcome.
>
> The *log-odds* is the logarithm of the odds.

Usually, logistic models make use of an indicator variable (which is a random variable that takes values 0 or 1). The log-odds of the value labeled as 1 is a combination of one or more independent variables. Let us see how this idea is formalized and how we use this.

Let $X$ be the input variables to a model and $Y$ a label with values $\{0, 1\}$ as we said before. The labels are distributed following a $Bern(p_i)$ distribution. That is:

$$y_i | x_{1,i}, \cdots, x_{n,i} \sim Bern(p_i)$$

We would like to have the conditional distribution $P(Y|X)$, that is, the distribution that models the probability of our label $Y$ given the input $X$.

Since $Y$ is an indicator variable, we have that $P(Y = 1) = E[Y]$ and, similarly, $P(Y = 1|X = x) = E[Y|X = x]$. Knowing this, there exists methods for estimating the regression function for the indicator variable using a smoother (Cosma, 2021). However this could be a bad idea because we can not assure that any smoother would return a value between 0 and 1. However, we can find a way to go around this problem.

Let us now assume $P(Y = 1|X = x) = p(x; \theta)$ for some function $p$ parametrized by $\theta$ and that the observations $x_i$ are independent from each other. Then, the conditional likelihood function is:

$$\prod_{i=1}^{n} P(Y = y_i | X = x_i) = \prod_{i=1}^{n} p(x_i, \theta)^{y_i} \left(1 - p(x_i; \theta)\right)^{(1-y_i)}. \tag{9}$$

We would like to find the $p(x;\theta)$, constraining it to be a probability, that is, to be between 0 and 1. To achieve this, we make use of the logistic transformation, that is, we consider:

$$\log \frac{p}{1-p}.$$

Let us set in the case of a single explanatory variable $x$. An idea could be to make this model a linear regressor. That is, solving

$$\log \frac{p(x;\theta)}{1-p(x;\theta)} = \beta_0 + \beta x,$$

which gives us the solution:

$$p(x;\beta) = \frac{exp(\beta_0+\beta x)}{1+exp(\beta_0+\beta x)} = \frac{1}{1+exp\,(-\beta_0+\beta x)},$$

which is called the logistic function. We have constructed the following definition:

**Definition 4.1.2.** The *standard logistic function* $\sigma : \mathbb{R} \to (0,1)$ is defined as follows:

$$\sigma(t) = \frac{1}{1+e^{-t}}$$

### 4.1.1 Fitting a Logistic Regression model

Consider now a generalized linear model, which we assume parametrized by a set of parameters $\theta$:

$$p(X,\theta) = \frac{1}{1+e^{-\theta^T X}} = P(Y=1|X;\theta).$$

WhereTherefore,

$$P(Y=0|X;\theta) = 1 - p(X,\theta).$$

Since $Y$ is an indicator variable, we have that

$$P(y|X;\theta) = p(X,\theta)^y (1-p(X,\theta))^{(1-y)}.$$

As we have seen in Equation (9), we have that the likelihood function in:

$$\begin{aligned} L(\theta|y;x) &= P(Y|X;\theta) \\ &= \prod_i P(y_i|x_i;\theta) \\ &= \prod_i p(x_i;\theta)_i^y (1-p(x_i,\theta))^{(1-y_i)}. \end{aligned}$$

If we consider now the log-likelihood function, the logarithm turns products into sums, so we obtain:

$$N^{-1} \log L(\theta|y;x) = N^{-1} \sum_{i=1} \log P(y_i|x_i;\theta),$$

which can be maximized using optimization techniques such as *gradient descent*, a very common technique in machine learning.

*Remark* 3. If we assume that the pairs $(x_i, y_i)$ are drawn uniformly from the underlying distribution, if we consider the the limit in $N$, we obtain:

$$\begin{aligned}
&\lim_{N\to+\infty} N^{-1} \sum_{i=1}^{N} \log P(y_i|x_i;\theta) \\
&= \sum_{x\in\mathcal{X}} \sum_{y\in\mathcal{Y}} P(X=x, Y=y) \log P(Y=y|X=x;\theta) \\
&= \sum_{x\in\mathcal{X}} \sum_{y\in\mathcal{Y}} P(X=x, Y=y) \left( -\log \frac{P(Y=y|X=x)}{P(Y=y|X=x;\theta)} + \log P(Y=y|X=x) \right) \\
&= -D_{KL}(Y||Y_\theta) - H(Y|X),
\end{aligned}$$

where $H(Y|X)$ is the conditional entropy, and in the first equality we have use the *Law of the Large Numbers*. This means that, intuitively, if we maximize the log-likelihood of a model, we are minimizing the KL-divergence of our model from the maximal entropy distribuition, which is sort of *searching for the model that makes the fewest assumptions in its parameters*.

## 4.2 FORMALIZATION OF THE NCE PROBLEM

Our problem now is to estimate the probability density function (p.d.f.) of some observed data. However, the data that we have available has been sampled in a specific way, let us present see how it is done.

Consider that a sample $X = \{x_1, \ldots, x_{T_d}\}$ has been observed, where each element has been sampled from a random variable. This data follows an unknown p.d.f. $P_d$, that we assume to belong to a parametrized family of functions, that is

$$P_d \in \{P_m(.;\theta)\}_\theta,$$

where $\theta$ is a vector of parameters. In other words

$$P_d(.) = P_m(.;\theta^*) \quad \text{for some } \theta^*.$$

Our problem now will be to find the $\theta^*$ that matches the distribution.

Any estimate $\hat{\theta}$ must meet the constraints that a normalized p.d.f. should satisfy.

**Definition 4.2.1.** Let $\hat{\theta}$ be a set of parameters and $P_m(.;\hat{\theta})$ a probability density function with parameters $\hat{\theta}$. We say that $P_m(.;\hat{\theta})$ is normalized if it satisfies:

$$\int P_m(u;\hat{\theta})du = 1, \quad \text{and} \quad P_m(.;\hat{\theta}) \geq 0.$$

If the constraints are satisfied for any $\theta$ in the set of parameters, we say that the model is normalized, and then we can use the maximum likelihood principle to estimate $\theta$.

Consider now some noisy data $Y$. Let us assume that the noisy data $Y$ is an i.i.d. sample $\{y_1, \ldots, Y_{T_n}\}$ of a random variable with p.d.f. $P_n$. The ratio $P_d/P_n$ of the density functions that generate $X$ and $Y$ respectively, can give us a relative description of the data $X$. That means, if we know the noisy

distribution $P_n$, then we can estimate the data distribution $P_d$ using the ratio that we have just mentioned. Shortly: if we know de differences between $X$ and $Y$ and we know the properties of $Y$, we can deduce the properties of $X$.

Our goal in *Noise Contrastive Estimation* (NCE) is to be able to discriminate between elements that have been sampled from the original data distribution $P_d$ and elements that have been sampled from the noise distributionIn order to discriminate between elements of $X$ and $Y$, it is needed to compare their properties. We will show that we can provide a relative description of $X$ in the form of an estimate of the ratio $P_d/P_n$.

Let $U = \{u_1, \cdots, u_{T_d+T_n}\}$ be the union of the sets $X$ and $Y$ that we mentioned before. We assign to each $u_t$ a binary class label, depending if it belongs to the original data $X$ or the noise data $Y$:

$$C_t(u_t) = \begin{cases} 1 & if\ u_t \in X \\ 0 & if\ u_t \in Y \end{cases}$$

With this labels, will now make use of logistic regression, where the posterior probabilities of the classes given the data are estimated. We know that the distribution of the data $P_d$ is unknown, we want to model the class conditional probability $P(.|C = 1)$ with $P_m(.;\theta)$. Note that $\theta$ may include a parameter for the normalization of the model, if it is not normalized. Hence, we have:

$$P(u|C = 1, \theta) = P_m(u;\theta), \qquad P(u|C = 0) = P_n(u).$$

Furthermore, since we know the numbers of examples of each of the $X$ and $Y$ sets, we also know that the prior probabilities are:

$$P(C = 1) = \frac{T_d}{T_d + T_n}, \qquad P(C = 0) = \frac{T_n}{T_d + T_n}.$$

We also have to consider that $P(u)$ can be decomposed as:

$$P(u) = P(C = 1)P_m(u;\theta) + P(C = 0)P_n(u) = \frac{T_d}{T_d + T_n}P_m(u;\theta) + \frac{T_n}{T_d + T_n}$$

Hence, if we rename the quotient of the probability of the negative class by the probability of the positive class $\nu = P(C = 0)/P(C = 1) = T_n/T_d$, the posterior probabilities for the positive class $C = 1$ is:

$$\begin{aligned} P(C = 1|u;\theta) &= \frac{P(u|C = 1;\theta)P(C = 1)}{P(u)} \\ &= \frac{P_m(u;\theta)P(C = 1)}{P(C = 1)P_m(u;\theta) + P(C = 0)P_n(u)} \\ &= \frac{P_m(u;\theta)}{P_m(u;\theta) + \nu P_n(u)} \end{aligned}$$

Where in the first equality we have used Bayes' rule. Similarly, the posterior probability for the negative class $C = 0$ is:

$$\begin{aligned} P(C=0|u;\theta) &= \frac{P(u|C=0;\theta)P(C=0)}{P(u)} \\ &= \frac{P_n(u)P(C=0)}{P(C=1)P_m(u;\theta) + P(C=0)P_n(u)} \\ &= \frac{P_n(u)P(C=0)}{P(C=1)\left(P_m(u;\theta) + \frac{P(C=0)}{P(C=1)}P_n(u)\right)} \\ &= \frac{P_n(u)\frac{P(C=0)}{P(C=1)}}{P_m(u;\theta) + \frac{P(C=0)}{P(C=1)}P_n(u)} \\ &= \frac{\nu P_n(u)}{P_m(u;\theta) + \nu P_n(u)}. \end{aligned}$$

We are in the conditions now to see how we use the logistic regression to obtain its conditional log-likelihood form. Let $G(.;\theta)$ be the log ratio between $P_m(.;\theta)$ and $P_n(.)$:

$$G(u;\theta) = \log \frac{P_m(u;\theta)}{P_n(u)} = \log P_m(u;\theta) - \log P_n(u). \tag{10}$$

Using this log-ratio, we obtain the following proposition. It is not a super interesting result, but we want to remark it since it connects the context with the logistic function.

**Proposition 7.** *Under the conditions that we have presented until this point, and naming $h(u;\theta) := P(C=1|u;\theta)$, we have that*

$$h(u;\theta) = r_\nu(G(u;\theta)),$$

*where*

$$r_\nu(u) = \frac{1}{1+\nu exp(-u)}, \tag{11}$$

*is the logistic function parametrized by $\nu$.*

*Proof.* Firstly, it is easy to see that

$$exp(-G(u;\theta)) = exp(-\log P_m(u;\theta) + \log P_n(u)) = \frac{P_n(u)}{P_m(u;\theta)}.$$

Using this, the proof is almost pretty straightforward:

$$\begin{aligned} h(u;\theta) &= r_\nu(G(u;\theta)) \\ &= \frac{1}{1+\nu\exp(-G(u;\theta))} \\ &= \frac{1}{1+\nu\frac{P_n(u)}{P_m(u;\theta)}} \\ &= \frac{1}{\frac{1}{P_m(u;\theta)}\left(P_m(u;\theta)+\nu P_n(u)\right)} \\ &= \frac{P_m(u;\theta)}{P_m(u;\theta)+\nu P_n(u)} \\ &= P(C=1|u;\theta), \end{aligned}$$

as we wanted to proof. □

Since the class labels $C_t$ are assumed Bernoulli distributed and independent, the conditional log-likelihood has the form:

$$\ell(\theta) = \sum_{t=1}^{T_d+T_n} C_t \log P(C_t=1|u_t;\theta) + (1-C_t)\log P(C_t=0|u_t;\theta). \tag{12}$$

Now, in the terms of the sumatory such that $u_t$ in $X$, we have that $u_t = x_t$ and, hence, $P(C_t=0|x_t;\theta)=0$. Because of this, we obtain that the term that adds to the sum in those specific $t$ adopt the form:

$$1\cdot\log P(C_t=1|u_t;\theta) = \log h(x_t;\theta).$$

Using the same argument for $t$ such that $u_t \in Y$, we obtain the following form of the log-likelihood in (12):

$$\ell(\theta) = \sum_{t=1}^{T_d}\log[h(x_t;\theta)] + \sum_{t=1}^{T_n}\log[1-h(y_t,\theta)]. \tag{13}$$

Now, optimizing $\ell(\theta)$ with respect to $\theta$ leads to an estimate $G(.;\hat{\theta})$ of the log-ratio $\log(P_d/P_n)$, so we get an approximate description of $X$ relative to $Y$ by optimizing (13).

*Remark* 4. If we consider $-\ell(\theta)$, this is known as the *cross entropy function*.

*Remark* 5. Here, we have achieved the estimation of a p.d.f. , which is an unsupervised (not labeled data) learning problem, using logistic regression, which is supervised learning (labeled data).

Now, if we consider $P_m^0(.;\alpha)$ an unnormalized (doest not integrate 1) model, we can add a normalization parameter to it in order to normalize it. We can consider

$$\log P_m(.;\theta) = \log P_m^0(.;\alpha) + c, \qquad \text{with } \theta=(\alpha,c).$$

With this model, a new estimator is defined. We consider $X$ as before and $Y$ an artificially generated set with $|Y| = T_n = \nu T_d$ independent observations

extracted from $P_n$, known. Then, the estimator is defined to be the argument $\hat{\theta}_T$ which maximizes

$$J_T(\theta) = \frac{1}{T_d}\left\{\sum_{t=1}^{T_d}\log[h(x_t;\theta)] + \sum_{t=1}^{T_n}\log[1-h(y_t;\theta)]\right\}.$$

We have to remark that in this case, we have fixed $\nu$ before $T_n$, so $T_n$ will increase as $T_d$ increases. Now, using the weak law of large numbers, $J_T(\theta) \to J$ in probability, where

$$J(\theta) = E\left[\log[h(x;\theta)]\right] + \nu E\left[\log[1-h(y;\theta)]\right].$$

Let us rename some terms before announcing a theorem. We want to see $J$ as a function of $\log P_m(.;\theta)$ instead of only $\theta$. In order to do this, let $f_m(.) = \log P_m(.;\theta)$, and consider

$$\tilde{J}(f_m) = E\left\{\log[r_\nu(f_m(x) - \log P_n(x))]\right\} + \nu E\left\{\log[1 - r_\nu(f_m(y) - \log P_n(y))]\right\}.$$

The following theorem states that the probability density function $P_d$ of the data can be found by maximizing $\tilde{J}$, that is, learning a nonparametric classifier in *infinite data*.

> **Theorem 4.1.** *The objective $\tilde{J}(f_m)$ achieves a maximum at $f_m = \log P_d$. Furthermore, there are not other extrema if the noise density $P_n$ is chosen such that it is nonzero whenever $P_d$ is nonzero.*

The proof of this theorem is beyond the scope of this work and will not be shown here. It can be found in Gutmann & Hyvarinen (n.d.).

# 5 THE INFONCE LOSS

We are now ready to connect the concepts of mutual information and generative models that we have presented. In unsupervised learning, it is a common strategy to predict future information and to try to find out if our predictions are correct. In *natural language processing*, for instance, representations are learned using neighbouring words (Mikolov *et al.*, 2013). In the field of computer vision, some studies have been able to predict color from grey-scale (Doersch *et al.*, 2016).

When we work with high-dimensional data, it is not useful to make use of an unimodal loss function to evaluate our model. If we did it like this, we would be assuming that there is only one peak in the distribution function and that it is actually similar to a Gaussian. This is not always true, so we can not assume it for our models. Generative models can be used for this purpose: they will model the relationships in the data $x$. However, they ignore the context $c$ in which the data $x$ is involved. As an easy example of this, an image contains thousands of bits of information, while the label that classifies the image contains much less information , say, 10 bits for 1024 categories. Because of this, modeling $P(x|c)$ might not be the best way to proceed if we want to obtain the real distribution that generates our data.

During the last few years, the representation learning problem has been approached using different machine learning frameworks. The most competitive ones have been self-supervised contrastive representation learning Oord *et al.* (2019); Tian *et al.* (2020); Hjelm *et al.* (2019); Gutmann & Hyvarinen (n.d.); Chen *et al.* (2020b); He *et al.* (2020a) using *contrastive losses*, and they have empirically outperformed other approaches.

In contrastive learning, different "views" of the same input are created. These are also called *positive samples*. Then, they are compared with *negative samples*, which are views created from an input that does not share information with the input of the positive sample. A very interesting idea would be to try and maximize the mutual information between positive samples and push apart the views taken from negative samples.

There are many ways of creating samples, both positive and negative, of an input. For instance:

- Randomly cropping different parts of an image. These would be examples of positive examples.
- Rotating or flipping images or crops of them would also be examples of positive samples.
- Taking different time-steps of a video would create positive samples.
- Selecting different parts of the same text would also be a positive example.

- Negative samples are created by applying one of the previous techniques to images that have nothing in common with the positive input.

In fact, if $v_1, v_2$ are two views of an input, we can think of the positive pairs as points coming from a joint distribution over the views $P(v_1, v_2)$, and negative samples coming from the product of the marginals $P(v_1)P(v_2)$, (Tian *et al.*, 2020).

It is important to find a way to determine how much shared information between the views is needed, in order to make the representations obtained good enough for any downstream task. Here is where the *InfoMin principle* is born.

> **Definition 5.1** (The InfoMin principle)**.** A good set of views are those that share the minimal information necessary to perform well at the downstream task.

Our goal here will be to seek for a way of extracting shared information between the context $c$ and the data $x$. Here is where we link the mutual information with representation learning. Remember that the mutual information of two random variables, say $x$ and $c$ in this case, is:

$$I(x, c) = \sum_{x,c} P(x, c) \log \frac{P(x|c)}{P(x)} \tag{14}$$

Maximizing the MI between $x$ and $c$, we extract the latent variables that the inputs have in common.

## 5.1 CONTRASTIVE PREDICTIVE CODING

We can apply these concepts in a concrete framework, presented firstly in Oord *et al.* (2019). Let us see what information is used and how it is treated in order to train a model that tries to obtain useful representations for downstream tasks.

In this section, if $x$ is an input signal for our network, $x_t$ will be the value of the input at instant $t$. We will also make reference to $x_{t+k}$, meaning that $x_{t+k}$ is $k$ steps ahead of time to $x_t$

Firstly, an *encoder* is used. An encoder is a model that, given an input $x$, provides a feature map or vector that holds the information that the input $x$ had.

So, we will use an encoder $g_{enc}$ that transforms the input sequence of observations $x_t$ to a sequence of latent representations

$$z_t = g_{enc}(x_t).$$

After we have obtained $z_t$, we use it as input of an autoregressive model, explained before, to produce a context latent representation:

$$c_t = g_{ar}(z_{\leq t}).$$

In this case, $c_t$ will summarize the information of $z_i$ for $i \leq t$. Following the argument that we gave before, predicting the future $x_{t+k}$ using only a

generative model (say $p_k(x_{t+k}|c)$) might not be correct, since we would be ignoring the context.

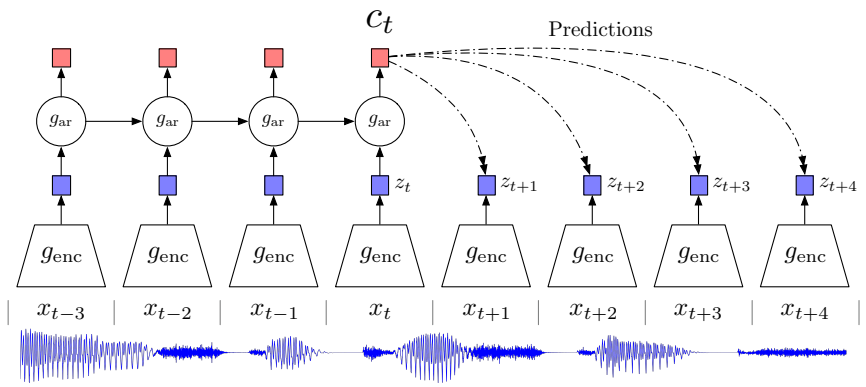



**Figure 15:** Image from (Oord *et al.*, 2019). Overview of Contrastive Predictive Coding framework using audio signal as input.

Let us see how we train the encoder $g_{enc}$ and the autoregressive model $g_{ar}$.

In Chapter 4, we gave the notions of the general idea of Noise Contrastive Estimation. Now, we can apply those ideas to a particular case, in which one of the subsets, say $X$ only has one element, and the other one has $N-1$ elements. We combine both sets in $X$ for the following argument.

Let $X = \{x_1, \dots, x_N\}$ be a set of $N$ random samples. $X$ will contain a positive sample taken from the distribution $P(x_{t+k}|c_t)$ and $N-1$ negative samples from the distribution proposed $P(x_{t+k})$. With this set, we would like to optimize the following loss function, which is an alternative expression of (13):

**Definition 5.1.1.** The loss $\mathcal{L}_N$ defined as

$$\mathcal{L}_N = -\underset{X}{E}\left[\log \frac{f_k(x_{t+k}, c_t)}{\sum_{x_j \in X} f_k(x_j, c_t)}\right], \tag{15}$$

is known as the *InfoNCE* (Information Noise Contrastive Estimation) loss. We have defined it this time for the particular case of this problem, but changing $f_k$ for a function depending on some parameters and $x_{t+k}$ and $c_t$ for positive and negative samples, this loss is generalized to a contrastive loss for any contrastive problem.

This loss also gives the bound in Proposition 6 the name of *InfoNCE Bound*.

It is clear that this loss is based in Noise Contrastive Estimation (Gutmann & Hyvarinen, n.d.).

Let us have a look at the *categorical cross-entropy* loss function:

$$\mathcal{L}(y, s) = -\sum_{i}^{C} y_i \log(s_i)$$

where $C$ is the number of possible classes in a classification problem, $y_i$ are the groundtruth of each class and $s_i$ is the score of each class.

As we remarked before, we can say that $\mathcal{L}_N$ is no more than the categorical cross-entropy of classifying the positive sample of $X$ correctly, with the argument of the logarithm being the prediction of the model. If we note with $[d = i]$ as an indicator of the sample $x_i$ being the positive sample in $X$, the optimal probability for this loss can be written as $P(d = i|X, c_t)$.

Now, the probability that $x_i$ was drawn from the conditional distribution $P(x_{t+k}|c_t)$ that has the context in account, rather than the proposal distribution $P(x_{t+k})$ that does not have $c_t$ in account, leads us to the following expression:

$$P(d = i|X, c_t) = \frac{\frac{P(x_i|c_t)}{P(x_i)}}{\sum_{j=1}^{N} \frac{P(x_j|c_t)}{P(x_j)}}.$$

This is the optimal case for (15).

In fact, if we denote $f(x_{t+k}, c_t)$ as the density ratio that preserves the mutual information between $x_{t+k}$ and $c_t$ in the mutual information definition (14), if $x_{t+k}$ is $k$ steps ahead on time respect to $x_t$, then

$$f_k(x_{t+k}, c_t) \propto \frac{P(x_{t+k}|c_t)}{P(x_{t+k})}, \tag{16}$$

where $\propto$ means that the member on the left is proportional to the member on the right. We can see that the optimal value $f_k(x_{t+k}, c_t)$ does not depend on $N - 1$, the number of negative samples in $X$. Using this density ratio, we are relieved from modeling the high dimensional distribution $x_{t_k}$. In Oord *et al.* (2019), for instance, the following log-bilinear model expression is used:

$$f_k(x_{t+k}, c_t) = exp(z_{t+k}^T W_k c_t).$$

In the proposed model, we can either use the representation given by the encoder $(z_t)$ or the representation given by the autoregressive model $(c_t)$ for downstream tasks. Clearly, the representation that aggregates information from past inputs will be more useful if more information about the context is needed. Furthermore, any type of models for the encoder and the autoregressive models can be used in this kind of framework.

It is clear how, using (15) we are using exactly the same function that we were using in Equation 7 in Chapter 3.2. This way, if we maximize this loss, we are also maximizing the mutual information between $x_{t+k}$ and the context $c_t$.

## 5.2 GOOD VIEWS FOR CONTRASTIVE LEARNING

We have presented a framework in which a set $X = \{x_1, \cdots, x_n\}$ contains a sample from the distribution $P(x_{t+k}, c_t)$ and the rest are samples from the distribution $P(x_{t+k})$. This samples are different views of the data.

The choice of the views affects the results in the downstream tasks (Tian *et al.*, 2020). The views will affect on the training, hence, it will affect to the

representations that are obtained. We would like to have some guarantees that the views that we are choosing provide us with good examples for our training. Let us formalize this idea.

Given two random variables $v_1, v_2$, our goal was to learn a function to discriminate the samples from the joint distribution and the product of the marginal distributions, resulting on a mutual information estimator between $v_1$ and $v_2$. In practice, $v_1$ and $v_2$ are two views of the same input $x$, using one of the methods that we mentioned in the introduction. We would like to have that, if $y$ is a downstream task, the mutual information between both the inputs and the downstream task, is the same as the mutual information between the input $x$ and the downstream task, i.e.:

$$I(v_1, y) = I(v_2, y) = I(x, y).$$

Also, we would like to remove the information that is not relevant for our downstream task. This is done by obtaining the pair of views $(v_1^*, v_2^*)$ such that the mutual information between them is the minimum of the mutual information between all the possible views $(v_i, v_j)$. Formally, that is obtaining

$$(v_1^*, v_2^*) = \min_{v_1, v_2} I(v_1, v_2),$$

These two ideas form the *InfoMin Principle* that we mentioned before in Definition 5.1.

Usually, the views are encoded using an encoder $f$, not having it to be the same for both views. We can say that $z_i = f_i(v_i)$ for $i \in \{1, 2\}$. If an encoder is sufficient, then it has to maintain the mutual information between the random variables after one of them has been encoded. More formally,

> **Definition 5.2.1.** We say an encoder $f_i$ of a view $v_i$, with $i \in \{1, 2\}$ is *sufficient* in the contrastive learning framework if, and only if it maintains the mutual information between the pairs $(v_i, v_j)$ and $(f_i(v_i), v_j)$ with $j \in \{1, 2\}$. That is
>
> $$I(v_i, v_j) = I(f_i(v_i), v_j).$$

This usually means that no information was lost in the process of encoding. We want to extract only the most essential information and do not learn the "extra" information between the views.

> **Definition 5.2.2.** We say that a sufficient encoder $f_i$ of $v_i$ is *minimal* if, and only if, the mutual information between $(f_i(v_i), v_j)$ is lesser than the mutual information between $(f(v_i), v_j)$ for any other sufficient encoder $f$. That is:
>
> $$I(f_i(v_i), v_j) \leq I(f(v_i), v_j) \quad \text{for all sufficient } \ f.$$

With these notions already presented, we would like to define what representations are good for a downstream task. We get to the following definition (Tian *et al.*, 2020):

> **Definition 5.2.3.** For a task $\mathcal{T}$, whose goal is to predict a label $y$ from the input data $x$, the optimal representation $z^*$ encoded from $x$, that is $z^* = f(x)$ for some encoder $f$, is the minimal sufficient statistic with respect to $y$.

This means that if we use $z^*$ to make a prediction using a machine learning model, we are using the same information that we would be using the whole input $x$. What is more, since we are following the InfoMin principles and we are dismissing all the non relevant information, $z^*$ provides with the smallest complexity.

**Proposition 8.** *Let $f_1, f_2$ be minimal sufficient encoders, and $\mathcal{T}$ be a downstream task with label $y$. Then, the optimal views $(v_1^*, v_2^*)$ from the data $x$ are the ones that have minimal mutual information*

$$(v_1^*, v_2^*) = \arg\min_{v_1, v_2} I(v_1, v_2),$$

*subject to to $I(v_1, y) = I(v_2, y) = I(x, y)$.*

*Given the optimal views $(v_1^*, v_2^*)$, the representation $z_1^*$ learned by contrastive learning is optimal for $\mathcal{T}$.*

The proof of this proposition is out of the scope of this work so no further information will be provided. The last statement of the Proposition 8 is a consequence of the minimality and sufficiency of $f_1$ and $f_2$.

This proposition carries the most important mathematical conclusion from this section. It will serve to prove sufficient conditions for views to be effective for contrastive learning, where effectiveness is measured as effectiveness in downstream tasks.

# 6 TRIPLET LOSSES

Contrastive learning exploits the idea of comparing the input $x$ with other different examples, either from the same class or from another class. It aims to produce closer representations for the examples of the same class and distant representations for elements of other classes.

We have been using the loss in Equation (15), which we built using noise contrastive estimation. However, this is not the only way of approaching this problem, and other types of losses have been used for similar purposes, forgetting the part of mutual information maximization and replacing it with *geometrical distance* optimization.

In this section, we will present *Triplet losses*, other kind of loss functions that also compare different views of the same input $x$.

## 6.1 FROM DEEP METRIC LEARNING TO TRIPLET LOSSES AND ITS GENERALIZATION

Distance metric learning also aims to learn an embedding representation of an input data $x$ that preserves the distance between similar data points close and also makes de distance between different datapoints far on the embedding space (Sohn, 2016).

Let us set the notation that we will use first. We will consider sets of triplets $(x, x^+, x^-)$ where:

- The element $x$ is an anchor point,
- The element $x^+$ is a positive instance,
- The element $x^-$ is a negative instance.

*Example 7.* Let us present a very simple example. If our input image is a cat, that would be the anchor $x$. Clearly, a positive instance would be an image of another cat or even the same cat seen from another perspective. A negative instance would be a photo of any other animal, in this case we use a dog.

Anchor Positive example Negative example

**Figure 16:** Example of an anchor $x$, a positive instance $x^+$ and a negative instance $x^-$. Images obtained from *Google*.

The main idea is to learn a representation of $x$, say $g(x)$, such that the distance of the representation of the input is closer in distance to the representation of the positive sample $x^+$ than the representation of the negative sample $x^-$. Using the norm[1], we can formally express that as follows:

$$\left\|g(x) - g(x^+)\right\|_2 \leq \left\|g(x) - g(x^-)\right\|_2,$$

for each triplet in the set.

Support-vector machines (SVMs) are supervised learning models used for classification or regression problems. They are one of the most robust prediction methods. They search for a hyperplane $h$ in high or infinite dimensional space that separates the data as much as possible, making use of *support vectors*, the datapoints that are closest to the hyperplane. If the data is linearly separable, we can select two hyperplanes $h_1, h_2$ that are parallel to $h$ and making the distance from them to $h$ as large as possible. That region is called the *margin*.

Coming back to our triplets problem, we also want to introduce a margin between the distances of the elements of the triplets, in order to separate positive examples from negative examples as much as possible. This way, we introduce a *margin* term $\alpha$, rewriting our last equation as follows:

$$\left\|g(x) - g(x^+)\right\|_2 + \alpha < \left\|g(x) - g(x^-)\right\|_2.$$

Using this inequality, we can define a hinge loss function for each triplet in the set:

$$\ell^\alpha(x, x^+, x^-) = \max\left(0, \left\|g(x) - g(x^+)\right\|_2^2 - \left\|g(x) - g(x^-)\right\|_2^2 + \alpha\right). \quad (17)$$

This loss has been defined for a single triplet. Now, we can define a global loss that accumulates the loss in Equation (17) using all the triplets in set.

1 A definition of the norm can be found on Appendix A, Definition A.1.

**Definition 6.1.1.** Given a set of triplets, each containing an anchor, a positive example and a negative example, $\mathcal{T} = \{(x_i, x_i^+, x_i^-)\}_{i\in\Lambda}$, we define a triplet loss as follows:

$$\mathcal{L}(x_i, x_i^+, x_i^-) = \sum_{i\in\Lambda} \ell^\alpha(x_i, x_i^+, x_i^-). \tag{18}$$

We use this loss to train models in order to improve the representations obtained.It would be interesting to present the model non-trivial metric to the learning algorithm. When the representation $g$ improves, this is harder to do, and this results in slow convergence and expensive data sampling methods.

## 6.2 GENERALIZATION OF TRIPLET LOSSES AND $N-$PAIRS LOSS

In a single evaluation of the loss function over a triplet during the learning process, we are comparing one positive sample to one negative sample. In practice, after looping over sufficiently many triplets, we expect the distance between positive examples and negative examples to be maximized. However, this will surely be a slow process if our dataset has many examples and also, in each step we will be separating the positive element from the specific negative element to which we are comparing it in that evaluation. Thus, the technique might be unstable (Sohn, 2016).

In order to fix this, a good idea would be to compare in each evaluation a positive sample with multiple negative samples, generalizing the case exposed before. This way, we would like the positive sample to increase its distance to *all* of the negative samples at the same time. Let us present a loss that generalizes the loss in Equation (18).

**Definition 6.2.1.** Let $x^+$ be a positive example of the anchor $x$, and consider the set $X^- = \{x_1^-, \cdots, x_{N-1}^-\}$ of $(N-1)$ negative samples. Given an encoder $g$, the $(N+1)-$tuplet loss is defined as follows:

$$\mathcal{L}_{(N+1)-\text{tuplet}}(x, x^+, X^-) = \log\left(1 + \sum_{i=1}^{N-1} \exp\left(g(x)^T g(x_i^-) - g(x)^T g(x^+)\right)\right) \tag{19}$$

*Remark* 6. If we consider the case $N = 2$, we have

$$\mathcal{L}_{(2+1)-\text{tuplet}}(x, x^+, x^-) = \log\left(1 + \exp\left(g(x)^T g(x^-) - g(x)^T g(x^+)\right)\right).$$

This expression is very similar to the one in Equation (17). In fact, if the norm in Equation (17) is unit and $g$ minimizes $\mathcal{L}_{(2+1)-\text{tuplet}}$, then it minimizes $\ell^\alpha$, and hence both losses are equivalent.

Applying the $(N+1)-$tuplet loss in deep metric learning is computationally expensive. Indeed, if we apply Stochastic Gradient Descent (SGD) with batch size $M$, then we have to evaluate $M \times (N+1)$ times our function $\ell^\alpha$ in each update. Because of this, if we increase $M$ and $N$, the number of evaluations grows quadratically. We would like to avoid this.

Consider the set of $N$ *pairs* of examples, with the constraint of each pair belonging to a different class, i.e. $X = \{(x_1, x_1^+), \cdots, (x_N, x_N^+)\}$ with $y_i \neq y_j$ for all $i \neq j$. We now build $N$ tuplets where each tuplet has all the positive samples and the $i-th$ anchor, that is:

$$\{S_i\}_{i=1}^N, \quad \text{where} \quad S_i = \{x_i, x_1^+, \cdots, x_N^+\}.$$

We can consider that each tuplet has $x_i$ as anchor, $x_i^+$ as positive example and $x_j^+$ for $j \neq i$ as negative samples, since they were all from different classes.

**Definition 6.2.2.** In the last conditions, we can define the *multi class N-pair loss* as follows:

$$\mathcal{L}_{N-pair-mc}\left(\{(x_i, x_i^+)\}_{i=1}^N\right) = \frac{1}{N}\sum_{i=1}^N \log\left(1 + \sum_{j\neq i}\exp\left(g(x_i)^T g(x_j^+) - g(x_i)^T g(x_i^+)\right)\right) \tag{20}$$

This way, we are combining the $(N+1)-$tuplet loss and the $N-$pair construction that we presented, enabling highly scalable training. This loss has empirically proved in to have advantages if we compare it to other variations of mini-batch methods (Sohn, 2016).

### 6.2.1 InfoNCE Bound as a triplet loss

The InfoNCE loss on Equation (15) has proved to be useful in representation learning. Let us consider a reformulation on it. Firstly, since $f_k$ was an exponential, we can also consider $e^f$ and remove the exponential from $f$, this is just notation. Now, in Poole *et al.* (2019) the InfoNCE bound in (8) is rewritten as follows:

$$I(X,Y) \geq E\left[\frac{1}{N}\sum_{i=1}^N \log \frac{e^{f(x_i,y_i)}}{\frac{1}{N}\sum_{j=1}^N e^{f(x_i,y_j)}}\right] \triangleq I_{NCE}(X,Y)$$

where we have just named the right hand side of the inequality as $I_{NCE}(X,Y)$. Now, we can transform it in the following way:

$$\begin{aligned} I_{NCE} &= E\left[\frac{1}{N}\sum_{i=1}^N \log \frac{e^{f(x_i,y_i)}}{\frac{1}{N}\sum_{j=1}^N e^{f(x_i,y_j)}}\right] \\ &= E\left[\frac{1}{N}\sum_{i=1}^N \log \frac{1}{\frac{1}{N}\sum_{j=1}^N e^{f(x_i,y_j)-f(x_i,y_i)}}\right] \\ &= E\left[-\frac{1}{N}\sum_{i=1}^N \log \frac{1}{N}\sum_{j=1}^N e^{f(x_i,y_j)-f(x_i,y_i)}\right] \end{aligned} \tag{21}$$

And now, we only have to use the basic properties of the logarithm to see that

$$
\begin{aligned}
(21) &= E\left[-\frac{1}{N}\left(\sum_{i=1}^{N}\log\frac{1}{N}+\sum_{i=1}^{N}\log\sum_{j=1}^{N}e^{f(x_i,y_j)-f(x_i,y_i)}\right)\right] \\
&= E\left[-\frac{1}{N}\left(N(\cancel{\log 1}-\log N)+\sum_{i=1}^{N}\log\left(1+\sum_{j\neq i}e^{f(x_i,y_j)-f(x_i,y_i)}\right)\right)\right] \\
&= E\left[-\frac{1}{N}\left(-N\log N\right)\right]+E\left[-\frac{1}{N}\sum_{i=1}^{N}\log\left(1+\sum_{j\neq i}e^{f(x_i,y_j)-f(x_i,y_i)}\right)\right] \\
&= \log N - E\left[\frac{1}{N}\sum_{i=1}^{N}\log\left(1+\sum_{j\neq i}e^{f(x_i,y_j)-f(x_i,y_i)}\right)\right] \qquad (22)
\end{aligned}
$$

In the particular case where $X, Y$ take values in the same space and $f$ has the particular form

$$f(x,y) = \phi(x)^T\phi(y),$$

for some $\phi$, Equation (22) is the same (up to constants and change of sign) as the expectation of the multi-class $N-$pair loss in Equation (20).

Hence, we have found an equivalence between representation learning by maximizing $I_{NCE}$ using a symmetric separable critic $f(x,y)$ and an encoder $g$ shared across views and using a multi class $N-$pair loss, which is a generalization of a triplet loss.

# Part III

# NEW FRAMEWORKS FOR REPRESENTATION LEARNING

# 7 SIMCLR

Until this point of the work, we have been presenting the theoretical basis of representation learning using contrastive learning. Previous approaches, such as the framework presented in Oord *et al.* (2019), use a generative approach as a part of the representation learning process. Although this can be beneficial at some points and, in fact, achieved the *state-of-art*[1] empirical results, we have to consider that generative models have some drawbacks.

Let us set in the case of learning representations of images to present a very simple example. In this case, generative models must *generate* each pixel on the image. This can be extremely computationally expensive.

Until now, we had been trying to minimize the loss in Equation (15), which we proved that maximizes a lower bound in the mutual information. However, some papers such as Chen *et al.* (2020b), Tschannen *et al.* (2020), suggest that it is unclear if the success of their methods is caused by the maximization of mutual information between the latent representations, or by the specific form that the constrastive loss has.

In fact, in Tschannen *et al.* (2020) they provide empirical proof for the loose connection between the success of the methods that use MI maximization and the utility of the MI maximization in practice. They also empirically proof that the encoder architecture can be more important than the estimator used to determine the MI.

Even with the empirically proved disconnection between MI maximization and representation quality, recent works that have used the loss function defined in Equation (15) have obtained state-of-art results in practice.

Both SimCLR and BYOL (the framework that we will present later), are examples of *siamese neural networks*. We remark it as a definition since it is important for the later explanation.

> **Definition 7.0.1.** A *siamese neural network* is a type of NN architecture that contains (typically) two NNs which have the same configuration, parameters and weights. The parameters are updated the same way across both networks.

These networks are usually used to find the similarity between the inputs by comparing produced feature vectors. They are more robust to class imbalance, nice to an ensemble with the best classifier (since the classifier is built on top of the representation extractors), and learn from semantic similarity.

1 *State-of-art* refers to the best results that have been achieved at some point of time.

## 7.1 SIMCLR FRAMEWORK

*SimCLR* (Chen *et al.*, 2020b) presents a framework that achieved state-of-art results when it was presented in July 2020. It also uses contrastive learning in an specific way that we will present later.

This framework learns representations by maximizing agreement between examples of the same input obtained by using data augmentation on the input example and a contrastive loss in the latent space.

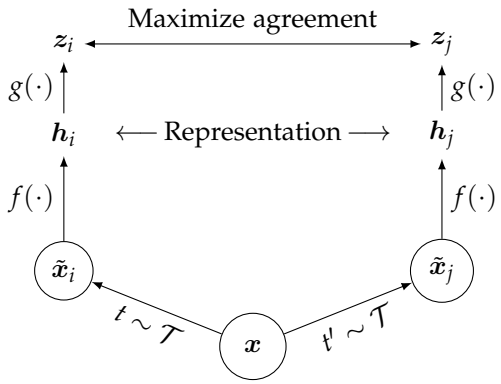



**Figure 17:** Figure obtained from Chen *et al.* (2020b). A simple framework for contrastive learning of visual representations.

The framework that is presented follows a linear structure. Figure 17 depicts it. Let us provide with deeper explanation. We will present the steps in a general way and later we will remark the specific considerations that were used in the implementation of the framework for the experiments.

The steps followed are:

1. Firstly, using the input $x$ and two augmentation functions $t, t' \in \mathcal{T}$, two different views $\tilde{x}_i, \tilde{x}_j$ are obtained using data augmentation. They are both sampled from the same family of data augmentations $\mathcal{T}$. They are *both* considered as positive views.
2. Secondly, a NN base encoder $f(\cdot)$ is used to extract representations for the two different views, obtaining $h_i = f(\tilde{x}_i)$, where $h_i \in \mathbb{R}^d$.
3. Then, a *small* neural network projection $g(\cdot)$ is used. This neural network maps the representations $h_i$ to the space where contrastive loss is applied. Hence, we obtain

$$z_i = g(h_i) = W^{(2)}\sigma(W^{(1)}h_i),$$

   where $\sigma$ is a nonlinear function and $W^{(i)}$ are the weights matrix.
4. Lastly, the contrastive loss is used for a contrastive prediction task. Using a set $\{\tilde{x}_k\}$ that includes a pair of positive examples $\tilde{x}_i, \tilde{x}_j$, the contrastive loss will (as we have already been doing theoretically) try to identify $\tilde{x}_j$ in the set for a given $\tilde{x}_i$. It is important to remark how the contrastive loss is used in this framework:

   a) A minibatch of $N$ samples is randomly taken from the training set.

b) Using the $N$ samples, we augment each pair to obtain $2N$ data points. The idea is, given a positive pair, use the other $2(N-1)$ as negative examples.

c) We define

$$sim(u, v) = \frac{u^T v}{\|u\| \, \|v\|},$$

the normalized dot product between $u$ and $v$. This function is also known as the *cosine similarity*. Then, the loss function for a positive pair of examples takes the form:

$$\ell_{i,j} = -\log \frac{exp(sim(z_i, z_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{k \neq i} exp(sim(z_i, z_k)/\tau)}, \tag{23}$$

where $\mathbb{1}_{k \neq i} \in \{0, 1\}$ produces 0 if $k = i$ and 1 elsewhere. $\tau$ is a temperature parameter that has to be adjusted.

d) The final loss is computed across all the positive pairs, both $(i, j)$ and $(j, i)$ in the minibatch.

$$\mathcal{L} = \frac{1}{2N} \sum_{k=1}^{2N} \left( \ell(2k-1, 2k) + \ell(2k, 2k-1) \right). \tag{24}$$

*Remark 7.* The loss in Equation (24) is just another formulation of the one in Equation (15), applied to this particular way of obtaining positive and negative views.

We can summarize all this steps in the following algorithm:

**Algorithm 1:** SimCLR's learning algorithm.

**input:** batch size $N$, temperature $\tau$, structure of $f$, $g$, $\mathcal{T}$.
**for** sampled minibatch $\{\boldsymbol{x}_k\}_{k=1}^{N}$ **do**
  **for all** $k \in \{1, \ldots, N\}$ **do**
    draw two augmentation functions $t \sim \mathcal{T}$, $t' \sim \mathcal{T}$
    # use augmentations
    $\tilde{\boldsymbol{x}}_{2k-1} = t(\boldsymbol{x}_k)$
    $\tilde{\boldsymbol{x}}_{2k} = t'(\boldsymbol{x}_k)$
    # use $f$
    $\boldsymbol{h}_{2k-1} = f(\tilde{\boldsymbol{x}}_{2k-1})$ # representation
    $\boldsymbol{h}_{2k} = f(\tilde{\boldsymbol{x}}_{2k})$ # representation
    # use $g$
    $\boldsymbol{z}_{2k-1} = g(\boldsymbol{h}_{2k-1})$ # projection
    $\boldsymbol{z}_{2k} = g(\boldsymbol{h}_{2k})$ # projection
  **end for**
  **for all** $i \in \{1, \ldots, 2N\}$ and $j \in \{1, \ldots, 2N\}$ **do**
    $s_{i,j} = \boldsymbol{z}_i^\top \boldsymbol{z}_j / (\|\boldsymbol{z}_i\| \|\boldsymbol{z}_j\|)$ # compute similarity
  **end for**
  update networks $f$ and $g$ to minimize $\mathcal{L}$ defined in Eq. (24)
**end for**
**return** encoder network $f(\cdot)$, and throw away $g(\cdot)$

**Figure 18:** Algorithm that summarizes the learning process that SimCLR follows.

Considerations:

1. In SimCLR, the data augmentation is applied sequentially, and using three techniques of image data augmentation that are very simple and common:
   a) Random cropping followed by resize to original size,
   b) Random color distortions, which consist of changing the value of certain amount of pixels randomly, like color jittering or changing from colored images to black and white images.
   c) Random Gaussian Blur: Gaussian Blur consists of convolving an image with a *Gaussian function*.
2. For the base encoder $f(\cdot)$, a *ResNet* network is used because of its simplicity.
3. For the projection head $g(\cdot)$, a MLP with one hidden layer is used, and *ReLU* is used as the activation function $\sigma$.

## 7.2 FINDINGS OF SIMCLR

Using the algorithm structure that we have presented, the experiments focused on finding what made the biggest impact on the representation learning task.

Intuitively, since we are comparing $2N$ images, either positive or negative, the batch size that we will use for the experiment should be as big as possible. Comparing a positive sample and trying to push it apart from as much negative samples as possible in the same iteration sounds like a good idea for the quality of our representations.

A few of the most relevant findings using SimCLR are:

- In the SimCLR framework, multiple data augmentations are used and it is empirically shown that this improves the contrastive prediction tasks that yield effective representations.
- The introduction of a nonlinear transformation that it can be learnt during the learning process. This linear transformation is between the representation and the contrastive loss, so before evaluating the loss function, the representation is applied the nonlinear function.
- The contrastive loss benefits from normalized embeddings and also from a temperature parameter that has to be adjusted.
- This loss also benefits from larger batch sizes and longer training, as well as from deeper and wider networks.

Later, *SimCLRv2* was presented (Chen *et al.*, 2020a), achieving a new state-of-art in semi-supervised learning on the ImageNet dataset. In this new framework, semi-supervised learning is use . This approach involves unsupervised learning followed by supervised fine-tuning[2]. After this stages that are similar to SimCLR, a final stage is added. It consists of *distillation* with unlabeled examples for refining the task-specific knowledge.

In the self-supervised (or unsupervised) part, the images are used without its labels, so that the representations of them that are obtained by the framework are not related to an specific task. Using this technique, Chen *et al.* (2020a) shows how using deeper and wider neural networks for both self-supervised pretraining and fine-tuning improves accuracy greatly. Also, the importance of the projection head $g(\cdot)$ is remarked in this new framework.

2 *Fine-Tuning* refers to the process of re-training certain parts of an already trained NN so that it focuses its learn on specific examples, such as a different dataset.

# 8 BOOSTRAP YOUR OWN LATENT

We have shown how contrastive methods rely on comparing different views of the same image with views of other images, considering the ones from the same image as positive and the rest as negative samples. This is why they are called *self-supervised* methods.

Self-supervised methods build upon the cross-view prediction framework, i.e., learning representations by predicting different views (or data augmentations) of the same image. This could lead the frameworks to collapsed representations, such as a constant representation for any view of the image can easily help us to identify objects, but it is useless for downstream tasks.

The methods presented earlier, which use contrastive learning, avoid this problem by trying to discriminate between positive and negative views, as we have already explained.

However, some papers (e.g. Caron *et al.* (2019)) have already raised the following question:

*Is the use of negative samples necessary to avoid collapsing?* .

This question is studied in Grill *et al.* (2020), paper that presents an algorithm that we will be deeply explaining.

The first solution that comes to mind to prevent the collapsing problem would be to use a randomly initialized network to produce the targets of the predictions. As it is probably expected, due to its randomness, it does not produce good representations for downstream tasks. Nonetheless, the representations obtained were empirically much better than initial fixed representation, so it could be interesting to refine this representation in order to make it better for the later tasks. This is the intuitive idea behind *Boostrap your own latent (BYOL)* (Grill *et al.*, 2020).

## 8.1 BYOL ALGORITHM

BYOL's algorithm has certain similarities with the SimCLR framework that we presented in Chapter 7. The goal of this framework is to learn a representation for an input. In this case, the representation will be noted as $y_\theta$.

For this purpose, two neural networks are used:

- An *online* network defined by a set of weights $\theta$.
- A *target* network, defined by a different set of weights $\xi$.

They both have the same structure, composed of three stages:

1. An encoder $f_\gamma$,
2. A projector $g_\gamma$,
3. A predictor $q_\gamma$,

where $\gamma \in \{\theta, \xi\}$. In the online network, despite their different name, the projector $g_\theta$ and the predictor $q_\theta$ have the same architecture.

*Remark* 8. The projector $g_\gamma$ is used because in SimCLR (Chen *et al.*, 2020b) is proven empirically that this projection improves the general performance of the framework.

The *difference* between them is that the target network provides the regression targets to train the online network, and its parameters $\xi$ are an exponential moving average[1] of the online parameters $\theta$. Mathematically, given a rate decay $\tau \in [0, 1]$, after each training step $\xi$ is updated as follows:

$$\xi \leftarrow \tau\xi + (1 - \tau)\theta$$

Having presented the networks and its structure, the following steps are followed in BYOL's framework:

1. The input that both networks receive is different, even if it comes from the same image. As in SimCLR, given an input image $x$ and two distributions of data augmentation for images, $\mathcal{T}, \mathcal{T}'$, two views are produced from $x$ to get $v = t(x), v' = t'(x)$ where $t \sim \mathcal{T}$ and $t' \sim \mathcal{T}'$.
2. Each produced view is passed to one of the networks. In particular, the first view $v$ is passed to the online network, and follows the next steps:
$$x \longmapsto v = t(x) \longmapsto y_\theta = f_\theta(v) \longmapsto z_\theta = g_\theta(y_\theta)$$
where $f_\theta, g_\theta$ are the ones that we mentioned before in the structure of each networks. This online network outputs $y_\theta$ and $z_\theta$.

   In a similar process, the the target network is passed the second view $v'$, which follows the next steps:
$$x \longmapsto v' = t'(x) \longmapsto y'_\xi = f_\xi(v') \longmapsto z'_\xi = g_\xi(y'_\xi)$$
where, again, $f_\xi, g_\xi$ are the ones mentioned before.
3. Then, using the online network, a prediction $q_\theta(z_\theta)$ is produced. Remark that the prediction is *only* applied to the online network.
4. Having $q_\theta(z_\theta)$ in the online network and $z'_\xi$ in the target network, they are both $\ell_2$-normalized to
$$\overline{q_\theta}(z_\theta) = \frac{q_\theta(z_\theta)}{\|q_\theta(z_\theta)\|} \quad \text{and} \quad \overline{z'_\xi} = \frac{z'_\xi}{\|z'_\xi\|}.$$
5. Now, we can define the mean squared error between the normalized prediction $\overline{q_\theta}(z_\theta)$ and the normalized projection $\overline{z'_\xi}$:
$$\mathcal{L}_{\theta,\xi} = \left\|\overline{q_\theta}(z_\theta) - \overline{z'_\xi}\right\|_2^2.$$

1 A moving average is a calculation to analyze data points by creating a series of averages in different subsets of the data.

6. If we stopped in the step 5, the framework would be asymmetric between the two networks, since the projection is performed in one of the views, $v$ but not in the other one, $v'$. To fix this, the process described is repeated except that now $v'$ is the input of the online network and $v$ is the input of the target network, producing a new loss $\tilde{\mathcal{L}}_{\theta,\xi}$. The final loss is computed as:

$$\mathcal{L}^{\text{BYOL}}_{\theta,\xi} = \mathcal{L}_{\theta,\xi} + \tilde{\mathcal{L}}_{\theta,\xi} \tag{25}$$

The loss in Equation (25) is the one that must be optimized stochastically. Furthermore, since we have expressed $\xi$ depends on $\theta$, if $\eta$ is the learning rate that we want to apply, the optimization problem can be expressed as follows:

$$\begin{cases} \theta \leftarrow \text{optimizer}\left(\theta, \nabla_\theta \mathcal{L}^{\text{BYOL}}_{\theta,\xi}, \eta\right), \\ \xi \leftarrow \tau\xi + (1-\tau)\theta \end{cases}$$

The framework can be summarized in the following figure:

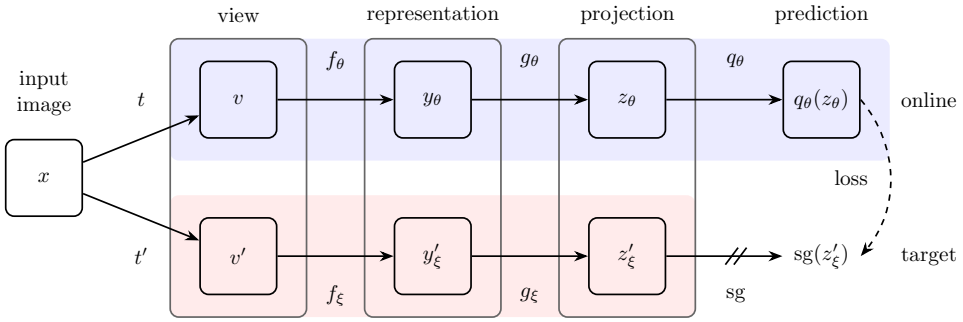



**Figure 19:** Image from (Grill *et al.*, 2020). Overview of Bootstrap Your Own Latent Framework.

The *sg (stop gradient)* indicates that, at that point, the gradient is not propagated back.

After the whole model has been fully trained, both the projection $g_\theta$ and the prediction $q_\theta$ are discarded, since what it is interesting for us is the representation of the input, and it is what it will be used in downstream tasks.

Some considerations have to be made about this framework:

1. About data augmentation: In BYOL,it is not important if the model learns about the histogram of the image, since we want ot keep any information captured by the target representation into its online network. In the original paper, it is empirically shown how BYOL is more robust to the choice of the image augmentations than SimCLR.
2. Since the weights of the target network $\xi$ depend on the weights of the online network $\theta$ by a parameter $\tau$, the first mentioned weights $\xi$ represent a delayed and stable version of $\theta$. Actually, if the decay rate $\tau$ is 1, we never update $\xi$, and if $\tau = 0$, then the weights of the target network are always being updated. This way, a trade-off is tablished

between updating $\xi$ too often or too slowly. Empirically, it has been shown that the most adequate values that yield to more stable results are $\tau \in [0.9, 0.999]$.

## 8.2 RESULTS OBTAINED BY THIS FRAMEWORK

In this framework, the impact of the batch size varies respect to the importance that it had in SimCLR. In contrast with BYOL's framework, in SimCLR we were using negative samples in our network to learn how to produce the representations. Because of this, BYOL's is expected to be more robust to smaller batch sizes.

It is empirically shown that the regularization of the weights is crutial in the self-supervised setting. The experiments in the paper showed that removing the weight decay regularization led to model divergence.

BYOL achieved the state-of-art performance results in the linear evaluation. This is, again, one of the most interesting parts since it helps us to evaluate how good the representations that our model create are.

| Method | Architecture | Params | Top-1 | Top 5 |
|---|---|---|---|---|
| SimCLR | ResNet-50(2×) | 94M | 74.2 | 92.0 |
| BYOL | ResNet-50(2×) | 94M | **77.4** | **93.6** |
| SimCLR | ResNet-50(4×) | 375M | 76.5 | 93.2 |
| BYOL | ResNet-50(4×) | 375M | **79.6** | **94.8** |

**Table 2:** Comparison between SimCLR and BYOL Top1 and Top5 accuracies using the same architectures on the ImageNet dataset.

BYOL also outperformed other models such as *MoCo* (He *et al.*, 2020b) and a second version of the Contrastive Predictive Coding framework (**?**), but they have not been included in the table since they are not deeply studied in this work.

# Part IV

# EXPERIMENTS

# 9 INTRODUCTION

In this chapter, we will explain the fundamentals and technologies that have been used for the experimentation. We will focus on three main aspects:

1. The used datasets.
2. The used libraries for the development of the code.
3. The used metrics to evaluate the obtained results.

The idea of the experimentation part is to test and compare the frameworks that we have presented in Chapters 7 and 8. Their architectures have already been explained, and the original code for both backbones has not been done by me.

This work will focus on testing how changing the training hyperparameters of the model affects the final results, since the original papers Chen *et al.* (2020b); Grill *et al.* (2020) already mention that using their structure, the results are affected by those hyperparameters, such as batch size, network depth or network width.

The implementations that have been used can be found in:

- SimCLR implementation: Official implementation from Google in `https://github.com/google-research/simclr/tree/master/tf2`
- BYOL implementation,: Official implementation from DeepMind, found in `https://github.com/deepmind/deepmind-research/tree/master/byol`.

At first, we tried to use the same library (Tensorflow) for both frameworks. However, finally it was decided that using the official implementation for BYOL, which was coded using `Jax` was better, since some pre-sets of this implementation were useful for our case study. Also, the idea is to make use of *Tensorboard*, a Tensorflow utility that helps with visualization and graph generation of the training and final results that can be used with both Jax and Tensorflow.

## 9.1 OBJECTIVES OF THE EXPERIMENTS

In the most ideal scenario, the main goal of our experiments should be to try to reproduce the results that the original papers have obtained and check if, under the conditions that are set, we obtain the same (or at least similar) results.

However, while carefully studying the frameworks proposed, we found that their results were obtained using the following computational resources:

- SimCLR's results were obtained using 128 TPU v3 cores in around 1.5 hours of training, using a batch size of 4096.
- BYOL's results were obtained using 512 Cloud TPU v3 cores in around 8 hours using the same batch size used in SimCLR experiments.

Approximately, the price of a single Cloud TPU V3 is 8.80$ per hour. If we wanted to use 128 TPUs, it would cost a total of $\sim$ 1126$ per hour of execution. We can not afford this pricing.

Because of this, we decided to set a different main goal of our experiments. Our goal will be to research what results can we obtain using limited resources. Using a single GPU for each experiment, we will investigate how the hyperparameters affect to the final results, in a dataset where the training times allow us to realize multiple training of the models.

## 9.2 LANGUAGE, HARDWARE AND BASIC LIBRARIES

The chosen language for this project was `Python`. There were other possibilities, such as `R`. However, since the majority of machine learning projects are developed in `Python`, our choice was easy.

When running training experiments in DL, the used hardware is one of the most determining factors for the results obtained by the models. There are many reasons for this, such as the time spent on the training a model or the amount of data that we can fit in the GPU's (which will be our case) or TPU's memory.

For the experiments of this project, the DECSAI[1]department of the University of Granada generously provided us access to a server that has a few NVIDIA GeForce RTX 3090. This model of GPU is one of the best in the market, having a *compute capability* of 8.6/10, rated by the NVIDIA company. IT also has a memory of $24GB$, which allows us to experiment with relatively large batch sizes using a single GPU and not having to parallelize the experiment.

To be able to use this GPUs in our experiments, three basic libraries have to be installed in the server:

1. `tensorflow-gpu`. This is a variant of the `tensorflow` library that was developed for using GPUs while using tensorflow.
2. `Jax`, a library used to automatically differentiate native `Python` and `NumPy` functions.
3. `CUDA 11.4`. CUDA is a parallel computing platform and API created by NVIDIA which allows to use a CUDA-enabled GPU for general purpose processing. In other words, CUDA is needed to be able to use the GPU in our `python` scripts.

These two libraries, along with other packages that are needed for creating graphics (`Matplotlib,Seaborn`) or computing metrics (`sklearn`) are installed using a `Conda` environment in our server's user.

1 The DECSAI's website is https://decsai.ugr.es/.

## 9.3 THE DATASETS

### 9.3.1 CIFAR10

The computational resources that we have for the experiment are limited. Due to this, we must fix a dataset that, having enough and representative examples, allows us to achieve feasible training time and successful results.

One of the ever most used dataset, which was also used in both SimCLR and BYOL papers, is CIFAR10 (Krizhevsky, n.d.). This dataset will be used to test the overall performance of our representation learning methods.

CIFAR10 contains 60.000 images divided in 10 classes, where each class contains 10.000 images. The size of the images is $32 \times 32 \times 3$, so the size of the images is not very large. This helps us to have faster trainings.

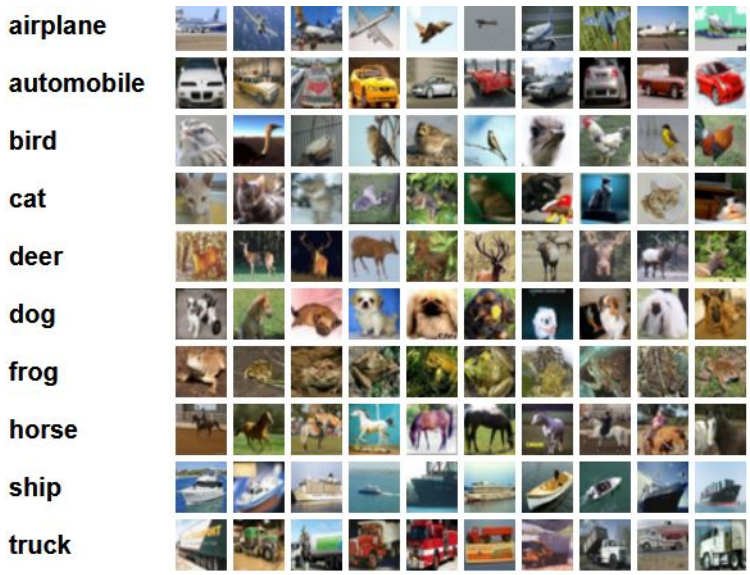



**Figure 20:** Ten examples of each class in the CIFAR10 dataset.

This dataset has 50.000 samples for training and 10.000 for test. The test batch contains the same number of examples of each of the 10.000 classes in the dataset, that is, it contains 1.000 examples of each class.

It is important to remark that the classes are *completely mutually exclusive*. That means that there is no overlap between the classes even if they have similar images, such as *Cars* and *Truck*, which are two of the classes of the dataset.

### 9.3.2 Imagenette

When we reached the part where we tried to reproduce the experiments of the BYOL framework, we discovered that the time for a single *pretraining* took almost a whole day to train for 100 epochs. Because of this, we decided to look for a solution that could make our trainings faster.

In the repository of the implementation, we found that the model could also be trained on *Imagenette*. This dataset is a subset of 10 easily classified classes from `Imagenet` (Russakovsky *et al.*, 2015), one of the biggest and most commonly used datasets. This subset can be obtained in different image sizes depending on our purposes: full size download, 320 px or 160 px. We use the 160 px version, consisting in approximately 9.000 images for train and 4.000 images for test.

Since the classes are easily classifiable, this dataset helps the project to test the frameworks in a friendly environment, similar to what CIFAR10 offers us, but using different classes.

## 9.4 TENSORFLOW

Tensorflow[2] is an open source library for developing machine learning frameworks.

It can be used for many tasks, but it focuses on training and inference of deep neural networks. It is used for both research and production at *Google*, since it was also developed by the *Google Brain* team for internal use. However, it was later released as open source.

Figure 21: Tensorflow logo.

The creation of new models is very simple, offering multiple abstraction levels. This is why it is suitable for our experiments. Also, the code is most of the times easily understandable.

There are a few ways to define a NN or a framework using tensorflow. The most classic one is defining a sequential model using `Keras`, a tensorflow API that defines layers of a neural network and helps with the implementation of simple NN structures. Let us see how to implement a very simple example of a NN with three *Dense* layers:

```
model = keras.Sequential(
    [
        layers.Dense(2, activation="relu", name="layer1"),
        layers.Dense(3, activation="relu", name="layer2"),
        layers.Dense(4, name="layer3"),
    ]
)
```

Another way of creating models using tensorflow is by defining a single step of training using `tf.GradientTape()` and then executing the single step multiple times in a loop. Using GradientTape , tensorflow performs automatic differentiation, which is needed for the minimization process. Let us see the simplest example, consider the function $f(x) = x^2$, and imagine that we want to obtain $f'(3)$. We can obtain it using `tf.GradientTape()` as follows:

2 Tensorflow documentation can be found at https://www.tensorflow.org/.

```
x = tf.constant(3.0)
with tf.GradientTape() as g:
  g.watch(x)
  y = x * x
dy_dx = g.gradient(y, x)
```

In our case, the gradient is obtained and the applied to the optimizer by using:

```
with tf.GradientTape() as tape:
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

### 9.4.1 Tensorboard

Tensorboard is a Tensorflow's visualization kit. It provides the visualization and tooling needed for machine learning experimentation. Among its more important utilities, we can find:

- Tracking and visualizing metrics (such as loss, accuracy, entropy) not only during the training but also when the training time has ended.
- Visualizing the model graph: ops and layers.
- Visualizing histograms of weights, biases and how tensors change during the training.
- Projecting high-dimensional data to a lower dimensional space.
- Displaying images,text and audio data.

Also, it is very easy to integrate with tensorflow. Actually, in most of the cases it is as simple as adding the following *callback* when we fit the model:

```
    tensorboard_callback = tf.keras.callbacks.TensorBoard
                        (log_dir=log_dir, histogram_freq=1)
    model.fit(x=x_train, y=y_train,
              epochs=5,
              validation_data=(x_test, y_test),
              # The added callback produces the magic!
              callbacks=[tensorboard_callback])
```

In our case, there are some differences, since we are not using the standard `fit` function to train our models. Because of this, we have to log the information that we have obtained in each step. To do this, we can use the `metrics` python package to group them (as it is done in the SimCLR code), or we can just directly save the information creating a *file writer* and writing the desired variables on this file. We do this in the modification that we have done to BYOL's original code as follows:

```
train_summary_writer = tf.summary.create_file_writer(args.log_dir)
with train_summary_writer.as_default():
    tf.summary.scalar('top_1_acc',float(acc),step=epoch)
    tf.summary.scalar('top_5_acc',float(top_5_acc),step=epoch)
    tf.summary.scalar('loss', float(losses[-1]), step=epoch)
```

## 9.5 METRICS

As we have seen, firstly, our models create a representation of the input image and then this representation is evaluated using a supervised linear head. This is the most interesting part, since we can see if the representation obtained was really useful for the classification task. We need to present the measures that we will use to measure how good the representations that we are producing are.

**Notation 1.** *We will address the* true positives *(the positive samples of a class classified correctly) as TP, the* true negatives *(the negative samples classified correctly) as TN, the* false positives *(the negative samples classified as positive ones, which is a mistake of our model) as FP and the false negatives (the positive samples classified incorrectly as negative samples) as FN.*

Using this notation, the main measure that we will be using is the *Accuracy*, which we know that can be expressed as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}.$$

Accuracy is a classic measure for classification models. It measures the percentage of correct labels that our models has obtained for the representation that has been introduced to the linear head as input. In our case, we will be distinguishing two cases of accuracy: measures that we will be using are the following:

1. *Top1* accuracy, which is the ordinary accuracy.
2. *Top5* accuracy, which measures if any of the 5 highest probability answers matches the true label. This is interesting because it tells us whether the model, if it did not classify an input correctly, it was close to doing it.

# 10 EXPERIMENTATION

We are now ready to perform the experiments. We will begin exploring SimCLR implementation and results, later we will explore BYOL's implementation and lastly we will compare them in order to see how BYOL tries to improve SimCLR and we will check if it is successful or not.

The code used for the experimentation, plus some files that contain the results, can be found on the Github repository made for this work.

## 10.1 SIMCLR EXPLORATION

We will perform an iterative exploration with this framework. We will explore a few range for a subset of the hyperparameters and then we will go deeper into some hyperparameters to try to obtain better results.

### 10.1.1 First approach

The first thing we did to experiment with this framework is to explore a wide range of hyperparameters to see which set of them performed better for us. The script that can be found in `code/SimCLR/run.py`.

What we did for this first exploration was to define a *parameter grid* and execute the whole framework using each combination of the parameters. The parameters that were firstly considered are:

1. `batch_size`. This is one of the most important parameters of SimCLR. In the original paper, it was proven that the higher this parameter, the better the results obtained for the linear classification. This parameter is also important since it has to adapt to our GPU's memory. The options that we have considered for the first experiments are: `batch_size = {512,1024}`.
2. `temperature`. The temperature parameter $\tau$ plays an important role in the individual loss seen in Equation (23). It is suggested to try values in the range $[0,1]$ and the one that had better performance for the original experiments was around 0.5, so we chose the following values: `temperature = {0.25,0.5,0.75,1}`.
3. `color_jitter_strength`. This parameter measures how hard is the color variation in the data augmentation. Previous results show that this parameter is important for the success of the network, so we provide with a big range of values. We include the following values: `color_jitter_strength = {0.25,0.5,0.75,1}`.

4. `epochs`. In the first stages of the experiments, we will always use 100 as standard number of epoch for the trainings. We will see later if more epochs are needed in order to obtain better results.

Using python and the python function `itertools.product` we straightforwardly generate all possible different combinations of unions of the parameters so we only have to append them to a general string and execute the `run.py` script mentioned before to obtain the results. In total, we obtain 32 possible combinations, so we will obtain 32 models.

The script was executed and took approximately 24 hours to train and evaluate all the different models, obtaining the results in Table 3.

| batch_size | temperature | color_jitter | regularization_loss | top_1_accuracy | top_5_accuracy | steps |
|---|---|---|---|---|---|---|
| **512** | **0.25** | **0.25** | **0.0093** | **0.833** | **0.994** | **9800** |
| | | 0.5 | 0.0089 | 0.832 | 0.993 | 9800 |
| | | 0.75 | 0.0086 | 0.831 | 0.994 | 9800 |
| | | 1.0 | 0.008 | 0.83 | 0.992 | 9800 |
| | 0.5 | 0.25 | 0.0136 | 0.819 | 0.993 | 9800 |
| | | 0.5 | 0.0124 | 0.822 | 0.993 | 9800 |
| | | 0.75 | 0.0121 | 0.821 | 0.993 | 9800 |
| | | 1.0 | 0.0118 | 0.817 | 0.992 | 9800 |
| | 0.75 | 0.25 | 0.0161 | 0.809 | 0.993 | 9800 |
| | | 0.5 | 0.015 | 0.812 | 0.993 | 9800 |
| | | 0.75 | 0.0141 | 0.805 | 0.99 | 9800 |
| | | 1.0 | 0.0137 | 0.793 | 0.99 | 9800 |
| | 1.0 | 0.25 | 0.017 | 0.798 | 0.99 | 9800 |
| | | 0.5 | 0.0163 | 0.797 | 0.99 | 9800 |
| | | 0.75 | 0.016 | 0.793 | 0.99 | 9800 |
| | | 1.0 | 0.0155 | 0.782 | 0.989 | 9800 |
| **1024** | **0.25** | 0.25 | 0.0103 | 0.836 | 0.993 | 4900 |
| | | 0.5 | 0.0097 | 0.839 | 0.995 | 4900 |
| | | **0.75** | **0.0093** | **0.841** | **0.995** | **4900** |
| | | 1.0 | 0.009 | 0.835 | 0.993 | 4900 |
| | 0.5 | 0.25 | 0.0149 | 0.822 | 0.993 | 4900 |
| | | 0.5 | 0.0133 | 0.827 | 0.994 | 4900 |
| | | 0.75 | 0.0132 | 0.826 | 0.994 | 4900 |
| | | 1.0 | 0.0128 | 0.82 | 0.993 | 4900 |
| | 0.75 | 0.25 | 0.0168 | 0.814 | 0.991 | 4900 |
| | | 0.5 | 0.0166 | 0.816 | 0.992 | 4900 |
| | | 0.75 | 0.0157 | 0.813 | 0.993 | 4900 |
| | | 1.0 | 0.0153 | 0.802 | 0.989 | 4900 |
| | 1.0 | 0.25 | 0.0175 | 0.806 | 0.991 | 4900 |
| | | 0.5 | 0.0174 | 0.802 | 0.992 | 4900 |
| | | 0.75 | 0.017 | 0.794 | 0.99 | 4900 |
| | | 1.0 | 0.0164 | 0.79 | 0.989 | 4900 |

**Table 3**: All results for first experiment using SimCLR using Resnet18.

We have to remark, for each of the two possible batch sizes, the best results obtained. These are:

| batch_size | temperature | color_jitter | regularization_loss | top_1_accuracy | top_5_accuracy | steps |
|---|---|---|---|---|---|---|
| 512 | 0.25 | 0.25 | 0.0093 | 0.833 | 0.994 | 9800 |
| 1024 | 0.25 | 0.75 | 0.0093 | **0.841** | 0.995 | 4900 |

**Table 4**: Best results for the grid search experiment with SimCLR.

*Remark* 9. On the original paper, the top 1 accuracy score reported for CIFAR10 in 100 epochs was $\sim$ 84.6% accuracy for batch size 512 and $\sim$ 85.1% accuracy score for 1024 batch size. It has to be remarked that these results were obtained using ResNet-50, larger version of ResNet. It is also reported that larger encoder architectures benefit the model, so this is an advantage they have at this point of the experiments . Having these aspects in account, and looking at the results obtained in our first experiment, we can say that this attempt was *successful*.

Let us make some observations. The clearest is that, since the second batch size doubles the first one, the training ends in half the steps. We can see that the temperature obtained in both cases is the same, and it is equal to 0.25. However, there is a big change in the color jitter parameter: while using 512 as batch size obtains the best result with the value of 0.25 for color jitter, using 1024 as batch size this value changes to 0.75, which implies stronger color changes on the data augmentation.

In general, having a look again at the Table 3 in Appendix **??**, we can see that:

- As it also happens in the experiments made on the original SimCLR paper, lower temperature $\tau$ parameters cause higher accuracies on the linear heads of our models.
- The models perform better when the `color_jitter` parameter is in the range $[0.5, 0.75]$. This means that, in general, a generous amount of color jittering to our picture is beneficial for the models.
- The *Top 5 accuracy* is above 99% in each model. This says that, since the models almost always give the correct label one of the 5 highest values, but they obtain the correct tag around 84% of the times, they might be creating some similar representations of the data for inputs that do not belong to the same class.

#### Observations about the batch size

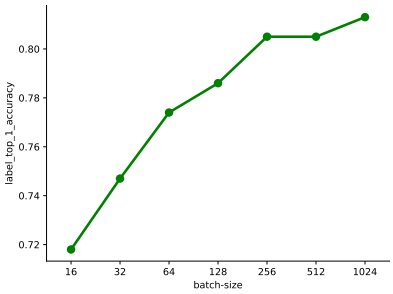
Although both models get to the same value of regularization loss, the model with a batch size of 1024 obtains a higher *top 1 accuracy*. This seems to follow the intuitive idea that we presented before: SimCLR benefits from bigger batch sizes since it allows a positive sample to be compared and pushed apart from negative samples. Ideally, we would keep pushing the batch size parameter forward, doubling it again to try to find out if the performance of the model keeps benefiting from bigger batch sizes. However, this requires either using multiple GPUs(or TPUs) working in parallel on the training or using GPUs with bigger memory, which we do not have access to.

Using the resources we have, the following experiment was performed: using the hyperparameters that we found out in our *GridSearch* to be the ones that achieve the best results except for the `batch-size`, we train models moving the batch size in the following range:

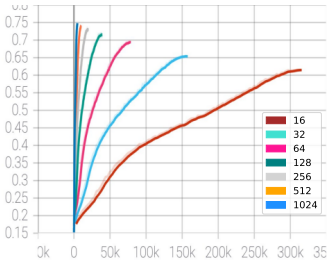
$$\texttt{batch-size} = \{16, 32, 64, 128, 256, 512, 1024\},$$

where the last two values had already been computed in the GridSearch so we do not have to repeat the training.

*Remark* 10. To avoid confusion, we have to remark that in the figures that will come in the next pages, all the models do not end the training in the same number of steps because the number of steps depends on the `batch_size`. That does not mean that the models were trained for a different number of epochs, and the figures are presented like this because Tensorboard provides the charts using the steps in the $x$ axis.




**(a)** Evolution of the top 1 accuracy with the batch size.




**(b)** Evolution curves of the accuracy with the steps taken in the training.

**Figure 22:** Results of the batch-size experiment.

Figure 22a shows the clear improvement of the Top 1 accuracy score when increasing the size of the batch that is used to compute the loss of our model. Increasing from 16 to 1024 gives an increase of more than 8% in top 1 accuracy, which is a lot. Further increase can be obtained if we keep increasing the batch size, but this is beyond our computation possibilities.

Figure 22b supports what we have just presented: not only with a smaller batch size we have to do thousands of extra steps in the training, but also we obtain much better accuracy performance of the linear head of the model.

### Observations of other hyperparameters

We have seen that in our first experiment, batch size has been really relevant in the final results. We would like to see if the rest of the parameters play such an important role as well.

As we have seen before, the parameters that we want to see how they affect the models are the `color_jitter_strength` and the `temperature` $\tau$ parameter. Let us study their impact on the model one by one.

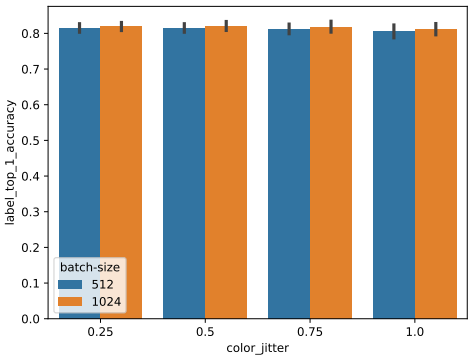



**Figure 23:** Accuracy score following the color jitter parameter and both batch sizes considered.

As we can see in Figure 23, the `color_jitter_strength` parameter does not have a huge impact on the performance of the linear head of the model. The black bars on the center of each orange/blue bar indicate the variance of the accuracy respect the rest of the parameters. Surely, there are centesimal differences between the different values of this parameter. However, more finetuning might be needed in order to make this parameter relevant for our model. The low influence may as well be caused by the small encoder network (ResNet 18), so we will explore if there are any changes when we change the encoder network later in the document.

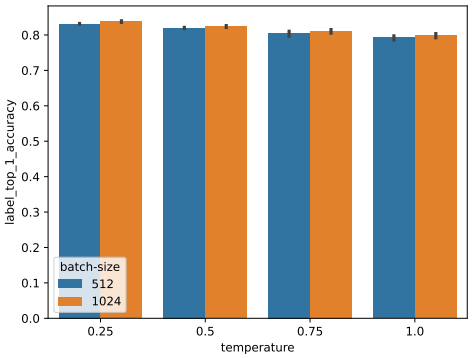



**Figure 24:** Accuracy score following the temperature and both batch sizes considered.

The parameter temperature, however, has a bigger impact on our framework's results. As we can see in Figure 24, when the temperature $\tau$ takes larger values, the accuracy obtained by the the linear head decreases. The decrease is approximately a 5% on average, so we can affirm that the influence of the parameter is important on the model.

In the original paper, the best results also occurred with low values for the temperature parameter. They even tested lower values for this parameter and got even better results. We will have this in account for the following experiments.

### 10.1.2 Going deeper on the encoder architecture

As we have mentioned before, original results prove that using wider and deeper architectures for the encoder on the SimCLR framework. Going deeper, however, requires more computational capabilities since more parameters have to be adjusted.

In the implementation used, we have the option of changing an input parameter in order to execute the experiment with a different architecture for the encoder. Specifically, we can pass `resnet_depth=50` as an argument to the `run.py` script to execute the train of the framework and the linear head finetuning and evaluation with the Resnet50 architecture. The architecture for this neural network is presented in Table 5.

| Layers | Output size | CIFAR10 |
|---|---|---|
| conv1 | $112 \times 112$ | $3 \times 3$, 64, stride 1 |
| conv2 | $56 \times 56$ | $3 \times 3$ max pool, stride 2 |
| | | $\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$ |
| conv3 | $28 \times 28$ | $\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$ |
| conv4 | $14 \times 14$ | $\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 6$ |
| conv5 | $7 \times 7$ | $\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$ |
| | $1 \times 1$ | $7 \times 7$ global average pool |
| | | $10 - d$ FC, softmax |
| Number of parameters | | 23.520.842 |

**Table 5:** Resnet50 architecture.

As we can see comparing this architecture with ResNet18 in Table 1, the number of parameters that the network has to train doubles the first one, so we expect to have less memory available on our GPU so the `batch_size` will have to be decreased to be able to run the trainings.

This is clearly an issue, since we have already empirically shown that our model benefits from taking larger values on this parameter. In this second stage of exploring the best parameters for SimCLR on CIFAR10, we used the following parameters set to perform our particular GridSearch:

- `batch_size`. In this second experiment, due to the limitations in the GPU memory, we have to reduce the range of this parameter to:

$$\texttt{batch-size} = \{16, 32, 64, 128, 256\}.$$

- `temperature`. We found out that, as it happens also to the original paper's experiments, lower temperature values help our representations to be better for downstream tasks. However, we decided to try again *higher* temperature values again, that is: temperature values in the set $\{0.5, 0.75\}$, to explore if the temperature parameter had any correlation with the encoder architecture.
- `color_jitter_stregth`. This time we focus on harder (higher) color jitter strengths, testing in the range $\{0.65, 0.75\}$, since in the first experiments the results obtained were better when this parameter was higher.

Using this parameters, our script to train and evaluate the models using each combination was executed. The results obtained can be checked in Table 6.

| batch_size | temperature | color_jitter | regularization_loss | label_top_1_accuracy | label_top_5_accuracy | global_step |
|---|---|---|---|---|---|---|
| 32 | 0.5 | 0.65 | 0.0243 | 0.822 | 0.992 | 157762 |
| | | 0.75 | 0.0238 | 0.814 | 0.992 | 157762 |
| 64 | 0.5 | 0.65 | 0.0223 | 0.835 | 0.995 | 78881 |
| | | 0.75 | 0.0222 | 0.833 | 0.994 | 78881 |
| | 0.75 | 0.25 | 0.021 | 0.838 | 0.995 | 78881 |
| | | 0.65 | 0.0242 | 0.828 | 0.993 | 78881 |
| | | 0.75 | 0.0243 | 0.822 | 0.993 | 78881 |
| 128 | 0.5 | 0.65 | 0.0225 | 0.844 | 0.994 | 39100 |
| | | 0.75 | 0.0222 | 0.842 | 0.994 | 39100 |
| | 0.75 | 0.65 | 0.0248 | 0.831 | 0.994 | 39100 |
| | | 0.75 | 0.024 | 0.829 | 0.994 | 39100 |
| **256** | **0.5** | 0.65 | 0.0235 | 0.846 | 0.995 | 19695 |
| | | **0.75** | **0.023** | **0.848** | **0.994** | **19695** |
| | 0.75 | 0.65 | 0.0255 | 0.832 | 0.994 | 19695 |
| | | 0.75 | 0.0262 | 0.833 | 0.994 | 19695 |

**Table 6**: All results for the SimCLR first experiment using Resnet50

We remark in Table 7 some of the most interesting results, comparing them with the ones obtained in the first experiment.

| resnet_depth | batch_size | temperature | color_jitter | regularization_loss | top_1_accuracy | top_5_accuracy | steps |
|---|---|---|---|---|---|---|---|
| 18 | 512 | 0.25 | 0.25 | 0.0093 | 0.833 | 0.994 | 9800 |
| | 1024 | 0.25 | 0.75 | 0.0093 | 0.841 | 0.995 | 4900 |
| **50** | 128 | 0.5 | 0.65 | 0.0225 | 0.844 | 0.994 | 39100 |
| | **256** | **0.5** | **0.75** | **0.023** | **0.848** | **994** | **19695** |

**Table 7**: Best results for the second grid search experiment with SimCLR.

As we can see, the results of the models that use ResNet50 as an encoder outperform the ones that use ResNet18 for a few hundredths in the top 1 accuracy score, and also they reach a lower value in the loss function. Since it is clear that the gain we obtain by increasing the encoder depth is minimal, we may think that it is not worth it for our model since the number of parameters that have to be trained and, thus, the time that the train takes is much higher. However, we must not forget a very important fact that we

observed in the first experiment: SimCLR benefits from bigger batch sizes. The results are comparing batch sizes 512 and 1024 in the first experiment with 128 and 256 in the second experiment (as we have already mentioned, the GPU memory does not allow us to use a bigger batch size when we increase the encoder depth).

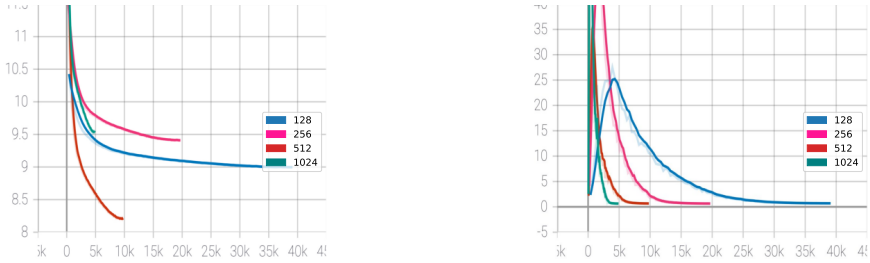
Taking this into account and although we can not empirically prove it, we state that when the batch size is increased as well as the encoder depth, the models with larger depth will outperform the models using ResNet18. Let us see some more details about the training.

In the official implementation of SimCLR, two losses are computed: a contrastive loss and a supervised loss. The total loss is computed as the sum of both:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{contrastive}} + \mathcal{L}_{\text{supervised}}.$$

*Remark* 11. The figures that will be shown during this chapter show charts of the evolution of certain values. The plots have been smoothed automatically by Tensorboard, using the default smoothing parameter: 0.6. Some *shadows* may appear in the charts, corresponding to the original values that the curve took before it was smoothed.

We can visualize how the loss evolve through the training using the Tensorboard output. The lines are smoothed, that is why sometimes there might be some *shadows* near the lines. The coefficient of smoothing is set as default ot 0.6.




**(a)** Contrastive loss in train

**(b)** Supervised loss in train

**Figure 25:** Charts of the losses during train in second experiment. On axis $x$ we have the number of steps and on $y$ axis the value of the loss.

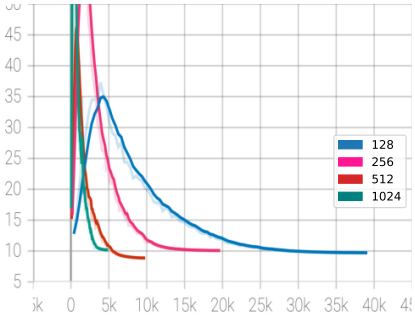



**Figure 26:** Total loss of the four remarked models in the second experiment. The axis are the same of the ones in Figure 25.

As we can see, although the loss is a sum of two components, as the steps/epochs get close to the end of the training, the supervised loss tends to zero. However, the contrastive loss remains in the range $[8, 9.5]$ approximately.

If we have a look at Figure 25b we can see that in all cases it reaches a minimum and although more steps of the experiments are done, the value of this loss does not get reduced. However, if we have a look at the contrastive loss in Figure 25a, we can see that in three out of the four models, the loss is still decreasing and if we increased the number of epochs,we state that we would probably obtain better results.

### 10.1.3 Gaussian blur in the image augmentations

We have also studied if adding *gaussian blur* as a part of the data augmentation process. This can be easily done by adding to the execution command the parameter `--use_blur=True` to the run script. As we have already mentioned, the types of transformations applied in the data augmentation process is very relevant for the model training. The quality of the representations strongly depends on the transformations used.

Our goal with adding gaussian blur to the experiment is testing if, in the case of this specific dataset (CIFAR10), adding gaussian blur as a part of the data augmentation improves the quality of the representations for the classification task performed later. It is very important to remark that we are testing this *only* for this dataset. The results obtained for this dataset do not have to be verified in general, since we know that most of the times, training a model in a dataset involves searching for the best hyperparameters for the specific dataset.

For this experiments, we decided to use wide range of hyperparameters and perform a third *Grid Search*. Concretely, we chose:

- `resnet_depth`. Although we already know and have proved that SimCLR benefits from larger ResNet depths, we wanted to see if adding data augmentation to the pretrain phase also improves the results of

the smaller ResNet depths. Because of this, we trained models on both *ResNet18* and *ResNet50*.

- `batch_size`. In this case, we fix a batch size of 256 for the ResNet50 trainings, since we already know that we can not fit bigger batch sizes in the GPU's memory. For the ResNet18 models, we test the same size that we use in ResNet50, and also add two bigger values to see how the model performs, so we use the set $\{256, 512, 1024\}$.
- `temperature`. In previous cases, the value that performed the best for us was 0.25. However, since in the original paper suggest to use 0.5 since it is the best parameter for *ImageNet*, we use this value as well, obtaining the set $\{0.25, 0.5\}$.
- `color_jitter_strength`. We use again the values $\{0.65, 0.75\}$ since we obtained before good results using these two values.

Using these parameters, we obtain the results that we depict in Table 8.

| resnet_depth | batch_size | temperature | color_jitter | regularization_loss | label_top_1_accuracy | label_top_5_accuracy | global_step |
|---|---|---|---|---|---|---|---|
| 18 | 256 | 0.25 | 0.65 | 0.0072 | 0.827 | 0.992 | 19695 |
| | | | 0.75 | 0.007 | 0.825 | 0.994 | |
| | | 0.5 | 0.65 | 0.0108 | 0.819 | 0.993 | |
| | | | 0.75 | 0.0108 | 0.82 | 0.993 | |
| | 512 | 0.25 | 0.65 | 0.0081 | 0.841 | 0.993 | 9800 |
| | | | 0.75 | 0.0081 | 0.837 | 0.994 | |
| | | 0.5 | 0.65 | 0.012 | 0.824 | 0.994 | |
| | | | 0.75 | 0.0118 | 0.823 | 0.994 | |
| | 1024 | 0.25 | 0.65 | 0.0093 | 0.846 | 0.994 | 4900 |
| | | | 0.75 | 0.0092 | 0.841 | 0.995 | |
| | | 0.5 | 0.65 | 0.0132 | 0.83 | 0.994 | |
| | | | 0.75 | 0.0131 | 0.833 | 0.994 | |
| **50** | **256** | **0.25** | **0.65** | **0.0228** | **0.862** | **0.997** | **19695** |
| | | | 0.75 | 0.0224 | 0.861 | 0.996 | |
| | | 0.5 | 0.65 | 0.0223 | 0.849 | 0.996 | |
| | | | 0.75 | 0.0215 | 0.851 | 0.995 | |

**Table 8:** Results for SimCLR using the best hyperparameters found in previous experiments, plus adding the Gaussian Blur to training preprocessing.

The models that obtained the best results in terms of Top 1 Accuracy are the following:

| resnet_depth | batch_size | temperature | color_jitter | regularization_loss | label_top_1_accuracy | label_top_5_accuracy | global_step |
|---|---|---|---|---|---|---|---|
| 18 | 1024 | 0.25 | 0.65 | 0.0093 | 0.846 | 0.994 | 4900 |
| **50** | **256** | **0.25** | **0.65** | **0.0228** | **0.862** | **0.997** | **19695** |

**Table 9:** Best results for the experiment of adding gaussian blur to data augmentation.

As we can see, the model that uses ResNet50 outperforms the one using a smaller ResNet architecture, as expected. Also, the temperature parameter that was obtaining the best results before, keeps doing it in this experiment, obtaining a final Top 1 accuracy of 0.862.

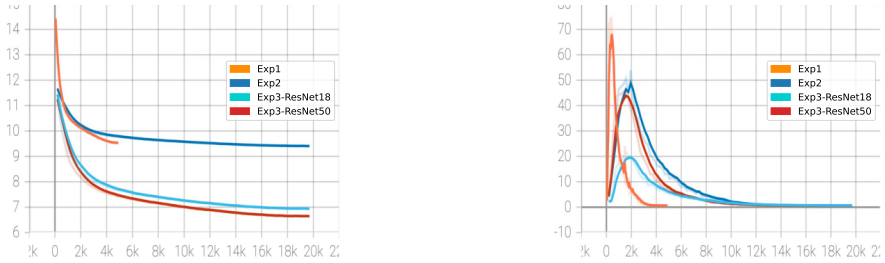
| Experiment | resnet_depth | batch_size | temperature | color_jitter | regularization_loss | label_top_1_accuracy | label_top_5_accuracy | global_step |
|---|---|---|---|---|---|---|---|---|
| 1st | 18 | 1024 | 0.25 | 0.75 | 0.0093 | 0.841 | 0.995 | 4900 |
| 2nd | 50 | 256 | 0.5 | 0.75 | 0.023 | 0.848 | 994 | 19695 |
| **3rd** | 18 | 1024 | 0.25 | 0.65 | 0.0093 | 0.846 | 0.994 | 4900 |
| | **50** | **256** | **0.25** | **0.65** | **0.0228** | **0.862** | **0.997** | **19695** |

**Table 10:** Comparison of the results of the third experiment with the best results of the two previous experiments.

As we can see , the model that uses ResNet18 in the 3rd experiment outperforms the model in the 1st experiment, which is a very similar model in terms of bath size, temperature parameter and color jitter. In fact, they reach the same loss value. However, this model of the third experiment get very slightly outperformed by the ResNet50 model of the second experiment. This emphasizes the importance of wider networks for the encoder.

However the best performing model is the one appearing on the last row in Table 10. This model uses gaussian blur and similar parameters to the best model of the second experiment, but increases the top 1 accuracy value for around 1.4%. This is a great improvement since we are already close to very high accuracies.

Also, recalling that 1st and 2nd experiment did not use blur and 3rd did, we can see that the models without blur need a stronger color jitter parameter and the model that uses blur uses a lower `color_jitter_strength` parameter. This may be a consequence of the blur, since we already know that applying a gaussian blur to an image smooths the color shifts in the image and, therefore, color jitter may have a lower impact on the representation learning.




**(a)** Contrastive loss in train

**(b)** Supervised loss in train

**Figure 27:** Charts of the losses during train in third experiment. On axis $x$ we have the number of steps and on $y$ axis the value of the loss.

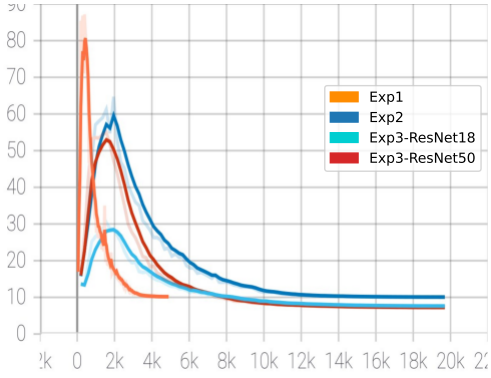



**Figure 28:** Total loss of the four remarked models in the third experiment. The axis are the same of the ones in Figure 27.

As we can see in Figure 27a, the models that use the smaller batch size (256) flatten their contrastive loss during the train. There is some margin for improvement in all cases, but probably this would not change the accuracy results notoriously, since the margin is narrow. This says that adding gaussian blur helps to finding a minimum in the loss function faster and converging to it with an appropriate learning rate ( even in our case where we have not finetuned the learning rate hyperparameter, we have used the default *adaptative* version). The model with the label *Exp1* is the best obtained from the first experiment, and as we mentioned in the second experiments, its loss values are still decreasing and probably could reach lower values (and, therefore, higher accuracies in classification) using more epochs for training.

Figure 27b shows that the loss of the linear head soon reaches values very close to zero, having no margin to improve. This, if the representations created were good for downstream tasks, was to be expected since we know that the already existing classifiers can perform very well (obtain high accuracies) in the dataset that we are using.

In general, having a look at the total loss shown in Figure 28 we can see the losses of all experiments are far from zero, but they have flattened the curve a lot, which means that probably with the used set of parameters to perform the grid search we will never reach *close to zero* values for this loss function.

### 10.1.4 Transfer Learning Bug

Transfer learning is the process where we use the weights of the models that we have obtained training the model in certain dataset and try to perform a small finetune and evaluation on another dataset, to check how what the model has learnt performs outside the training dataset. In this case, our we have trained this framework using CIFAR10, we would like to finetune the weights for Imagenette.

The official Google's SimCLR repository has the tools for this. Theoretically, the task was as simple as executing the `run.py` script passing, `--train_mode=finetune` along with the number of layers to finetune, and `--checkpoint=model_dir`, with the directory where the weights that we want to use are.

However, when I tried to run this transfer learning experiment, I found that there were execution errors that probably should not be happening. I decided to contact the authors of the code by adding an *issue*[1]. One of the main authors of the paper and the code answered that, indeed, I had found a **bug** on the code that Google's researchers implemented.

The bug appears when the transfer learning is tried to be done in dataset with different image sizes. In the image preprocessing process for evaluation, an image of bigger size than expected is received, obtaining the following error

**Listing 10.1:** Error obtained in transfer learning.

1 The issue can be found and read through the original repository, or in https://github.com/google-research/simclr/issues/163.

```
(0) Invalid argument:  Input to reshape is a tensor with 90720

values, but the requested shape has 3072
    [[node Reshape (defined at run.py:395) ]]
```

The code authors from Google tried to give us a quick way to fix this, which consisted of resizing the input image to $32 \times 32 \times 3$ (the size that we used to train the framework in CIFAR10) right before passing the image to the model. However, even with this quick fix provided by the collaborators, the error persisted since we found that in some place some column/rows were deleted so the sizes of the tensors (images) and the weights were not compatible.

### 10.1.5 SimCLR Conclusions

Once the experiments with this framework have finished, we can remark the most important facts that we can observe in the results of the executions. We can conclude that:

- In general, the choice of `temperature` and `color_jitter_strength` parameters seems relevant and has high impact on the quality of the representations obtained. However, if we had to choose between one of them, we have observed that the `temperature` seems to have a major importance on the final results, since using lower values for this parameter leads to reasonably large increases on the accuracy metrics and decreases on the loss functions. All this considered, we state that it is clear that using lower temperature parameters (around 0.25) is the best choice, but the `color_jitter_strength` parameter is dependent on all the other parameters.
- It is crucial to use `batch_sizes` as larges as possible. It has been empirically proved that the correlation between a higher batch size and higher accuracy score is high. In our case, the biggest value that we can use with our resources is (for the ResNet50 architecture in the encoder) 256.
- The encoder architecture is key for the quality of the representations. Even if we have to train more parameters, the train times are longer and the batch sizes used are smaller, the performance of the classification tasks is improved by a wide margin.
- For the CIFAR10 case, adding `gaussian blur` to the preprocessing and data augmentation stage is beneficial. It leads to higher higher accuracy scores in the classification tasks, which makes us think that the representations created by the encoder are better for downstream tasks.

| resnet_depth | batch_size | temperature | color_jitter | label_top_1_accuracy | label_top_5_accuracy |
|---|---|---|---|---|---|
| 50 | 256 | 0.25 | 0.65 | 0.862 | 0.997 |

**Table 11:** Conclusion results in SimCLR.

With all this considerations, the best hyperparameters found are the ones presented in Table 11. As we can see, we achieve 86.2% as top 1 accuracy best score. This result is far from the 95.3% reported in the original paper. However, in the paper the models were pretrained in ImageNet and used ResNet50 4× so, in comparison, we can state that we got as close as we could using random initialized weights (instead of pretrained in ImageNet) and smaller encoder architecture.

## 10.2 BYOL EXPLORATION

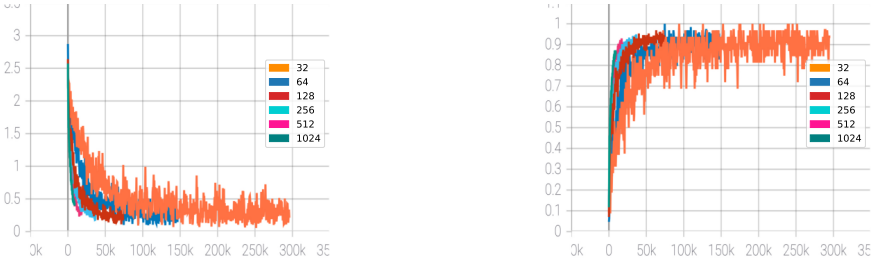
Using the same iterative process that we have followed to explore SimCLR, we are going to train the models of this framework using different hyperparameters and we are going to see what results we obtain.

In this case, the number of hyperparameters that we are able to explore is drastically reduced, we will focus on two hyperparameters that showed to be very important in the previous framework. Specifically, we will try to determine how the following hyperparameters affect the results:

- `batch_size`. This parameter was very important in the SimCLR framework. However, in BYOL's original paper (Grill *et al.*, 2020), it is empirically shown that this hyperparameter does not have the same relevance in this new framework. We want to check this ourselves in the considered dataset.
- `resnet_depth`. Since it is repeated in many cases, the depth and width of the used networks affect the results in a directly proportional way: the wider and deeper networks, the better representations we obtain. Even even a new version of SimCLR: (*SimCLRv2*) emphasizing this idea (Chen *et al.*, 2020a). We want to see if this hypothesis is also fulfilled in the used dataset.

At first, we wanted to give both models the same treat and try to perform the grid search algorithm with the same amount of hyperparameters. However, in a first attempt to train BYOL in CIFAR10 we found out that the training times were much longer than SimCLR training times. A single execution of the pretraining and evaluation process would take around 15 hours to finish, which was more time than we could afford to spend for each model. Due to this, we decided to change the dataset to smaller one: Imagenette.

Before introducing the results obtained, a little observation has to be made. When we configured the tensorboard logs, we fixed a log per epoch. Due of this and probably to the value of the learning rate being bigger than it should be, we can see in Figure 29 that the lines have lots of ups and downs even if they are converging in time. Because of this we have to increase the smoothing value from 0.6 to 1, which will result in showing big shadows on our charts, indicating the *real values* that the lines would take. We use the smoothed curves for our comparisons, since they are more representative of how the individual model would perform on average.




**(a)** Not smoothed supervised loss

**(b)** Not smoothed top 1 accuracy

**Figure 29:** Examples of not smoothed charts in BYOL's experiments.

### 10.2.1 Pre-sets on data augmentation and implementation possible improvements

Data augmentation hyperparameters are not finetuned in this framework for a few reasons, including the long execution times. Despite this, since the importance of data augmentation has been recalled many times, we will explain the choice of hyperparameters that are pre-fixed in the implementation that we have chosen to run our experiments.

The first augmented view is applied the following transformations:

1. *Random Flip*, selecting a random amount of images of the batch used and randomly flipping them.
2. *Color Transformation*. This transformation is set to be applied with a probability of 1, i.e., to always be applied. It adjusts the brightness, contrast and saturation of the image. It also has a probability of 0.8 to apply color jitter and 0.2 of transforming to grayscale. All this operations are shuffled to be applied in a random order.
3. *Gaussian blur*. This transformation is also applied always to the image, using a $kernel\_size = \frac{\text{image size}}{\text{blur\_divider}}$, where we know that the image size is 32 and `blur_divider` is pre-fixed to 10. Also, the standard deviations are in the range $[0.1, 2]$.

The second view follows the same process for the augmentations, but it is also sometimes (with probability 0.2) applied a *solarization*. This technique consists of reversing the tone of the whole or part of the image.

The original repository provides with the steps that can be followed to change the pre-sets of their code. However, we want to remark that the implementation could be generalized in some ways:

1. Firstly, this implementation has the hyperparameters for the data augmentation process are fixed, as we have just mentioned. This could be improved by adding more FLAGS or parameters to the `main_loop.py` script. This would not be a huge cost for the implementation and would lead to a wider range of experimentation possibilities.
2. Along with the first change, the same could be done with not only the ResNet architecture, but also with the used dataset.

This all can be changed manually to be able to run any experiment by modifying the source configurations provided. We also could have coded this generalization ourselves, but it was decided to focus on the experiments mentioned at the beginning of this section.

### 10.2.2 First experiment: batch size influence

As we have stated, we would like to see how the batch size influences the representations obtained with this framework. Beforehand, and recalling that the loss we use in this framework is written as follows

$$\mathcal{L}_{\theta,\xi} = \left\| \overline{q_\theta}(z_\theta) - \overline{z'_\xi} \right\|_2^2 \quad ,$$

we observe that, in contrast to SimCLR where the loss function compared $2N$ elements, there should be no correlation between the batch size and the loss function since only two elements are used to compute the loss.

In this first experiment, we will fix ResNet18 for the encoder architecture, and will use the following values for the batch size parameter:

$$\text{batch\_size} = \{32, 64, 128, 256, 512, 1024\}$$

More values can not be tested since, if we try to use the following power of two, the execution fails indicating that the memory of the GPU cannot allocate the whole batch. Recall that for the experiment we also use the hyperparameters presented in Subsection 10.2.1.

If we run the experiments, we obtain the results that we fully present in Table 13.

| batch_size | regularization_loss | label_top_1_accuracy | label_top_5_accuracy | global_step |
|---|---|---|---|---|
| 32 | 0.723 | 0.875 | 1 | 295900 |
| **64** | **0.381** | **0.968** | **1** | **148000** |
| 128 | 0.436 | 0.929 | 0.976 | 739800 |
| 256 | 0.453 | 0.929 | 0.996 | 369900 |
| 512 | 0.520 | 0.916 | 0.994 | 184900 |
| 1024 | 0.752 | 0.852 | 0.983 | 924700 |

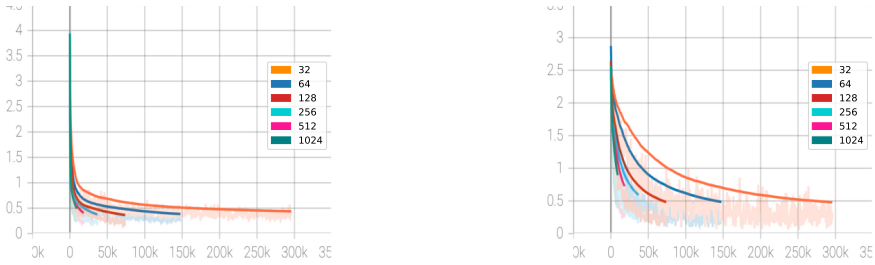
**Table 12:** All results for BYOL's experiment on the influence of batch size.

We present here the two most important models obtained:

| batch_size | regularization_loss | label_top_1_accuracy | label_top_5_accuracy | global_step |
|---|---|---|---|---|
| 32 | 0.723 | 0.875 | 1 | 295900 |
| **64** | **0.381** | **0.968** | **1** | **148000** |

**Table 13:** Most important results for BYOL's experiment on the influence of batch size.

The models shown in Table 13 are the ones that obtain the highest value in top 1 accuracy score, which will be using to select the best model found. As we can see, in this case the *best* models are the ones that use the smallest




**(a)** Representation loss in train

**(b)** Supervised loss in train

**Figure 30:** Charts on the losses during train in the first experiment with BYOL framework. On axis $x$ we have the number of steps and on $y$ axis the value of the loss.

batch sizes, reaffirming the theory that batch size does not matter in this framework as much as it did in SimCLR.

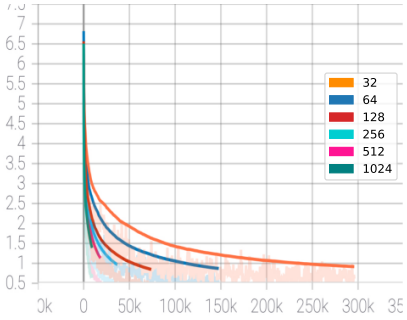



**Figure 31:** Total loss of all the models for the first experiment in BYOL. The axis are the same of the ones in Figure 30.

As we can see in Figure 31, the total loss (computed as the sum of both representation and supervised losses), achieves much lower values compared to SimCLR values: all of them are in the range $[0.7, 1.4]$. This causes higher top 1 accuracy in classification. This is not surprising since this dataset is much smaller in terms of number of examples, so the model has facilities to learn from all the examples and possibly produce overfitting in the dataset.

Figures 30a and 30b show that the models with lower batch sizes flatten the curve more than the ones with lower values on this parameter. If we combine this information with the fact that the models with smaller batch size perform better in classification, we can state that the number of steps is relevant for an experiment like this, and more epochs should be given to the models with bigger batch sizes in order to let them train for a similar number of steps and verify if the models with smaller batch size still achieve better results in the downstream task than the ones with bigger batch size.

### 10.2.3 Second experiment: ResNet depth influence

In the original paper, it is reported that BYOL benefits from larger encoder architectures as it happens in SimCLR. We want to experiment using the same architecture that we used in the last experiment with SimCLR, ResNet50, and test if we achieve better performance on the linear evaluation when we go deeper in the encoder architecture.

To be able to do this, we modify the `configs/byol.py` script , changing the following variable to `encoder_class='ResNet50'`. We run the main loop again and save the results using tensorboard records. All the results obtained are recorded in Table 14.

| batch_size | regularization_loss | label_top_1_accuracy | label_top_5_accuracy | global_step |
|---|---|---|---|---|
| 32 | 0.564 | 0.937 | 0.9688 | 295900 |
| 64 | 0.484 | 0.953 | 1 | 148000 |
| **128** | **0.2747** | **1** | **1** | **739800** |
| 256 | 0.276 | 0.968 | 1 | 369900 |

**Table 14:** All results for BYOL's experiment on the influence of the encoder architecture.

| ResNet | batch_size | regularization_loss | label_top_1_accuracy | label_top_5_accuracy | global_step |
|---|---|---|---|---|---|
| 18 | 32 | 0.723 | 0.875 | 1 | 295900 |
| | 64 | 0.381 | 0.968 | 1 | 148000 |
| **50** | **128** | **0.2747** | **1** | **1** | **739800** |
| | 256 | 0.276 | 0.968 | 1 | 369900 |

**Table 15:** Results of the best models in both first and second experiments with BYOL.

In Table 15 we report the best two models of each of the experiments performed with BYOL. As we can see, the model that obtains the highest top 1 accuracy is the one that uses ResNet50 as the encoder, that is, the one with deeper encoder architecture.

We also have to remark that the best model achieves perfect accuracy on the classification task, which is really surprising. The representations that the model is creating for the Imagenette dataset are very accurate, so the linear head can classify all the elements in the test set correctly. Also, as we can see, all the models obtain 100% top 5 accuracy, which means that even if the prediction that the linear head does is not correct, it is always close to be correct, so this informs us that our models advance towards learning representations that contain a lot of information about the original input.


**(a)** Representation loss in train

**(b)** Supervised loss in train

**Figure 32:** Charts on the losses during train in the second experiment with BYOL framework. On axis $x$ we have the number of steps and on $y$ axis the value of the loss.

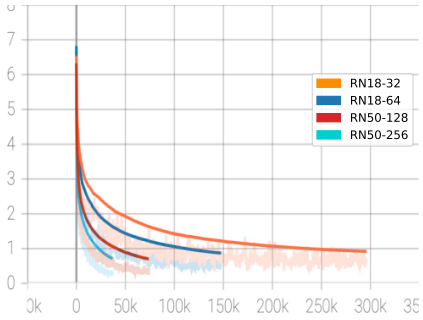



**Figure 33:** Total loss of the models that we are comparing in the second experiment done using BYOL. The axis are the same of the ones in Figure 32.

Observing Figure 32a, we can see that the selected as best models reach really low values on the representation loss. This is consistent with the results that we later obtain using these representations for the classification task with the linear head. The model with the biggest batch size performs the fewest number of steps, and it does not achieve to completely flatten the loss curve. However, this is not bad news, since it indicates that this model can still improve in the representation learning task and, therefore, improve the final accuracy score on the classification task.

In Figure 32b we can see that, in the supervised part, the improvement can come from any of the models, since none of them completely flattens the loss curve in the amount of steps that we have set for them. Adding the two losses we obtain the total loss represented in Figure 33. If we remember the first experiment, the range for the total loss was $[0.7, 1.4]$, in this case the total loss values are reduced to the range $[0.4, 0.7]$ which indicates lower losses in general. This is also consistent with the better accuracy scores in the linear evaluation.

### 10.2.4 Conclusions about BYOL

With the two experiments that we have done using BYOL, we can conclude the following:

- Unlike SimCLR, `batch_size` is not a key hyperparameter for the performance of the framework. In fact, one of the smallest batch size values obtained the best result in the first experiment.
- As in SimCLR, the encoder depth influences the final results on the classification task and, therefore, we can assume that influences the quality of the representations for any downstream task performed.
- The accuracies that this framework obtains are quite higher compared to SimCLR's scores. Recall that both models were pretrained from zero, but trained in different datasets. However, the different datasets should not be problematic since both of them have similar properties (in terms of number of classes and separability of the classes) and the main difference is the number of examples.

Recall that we achieved 100% top 1 accuracy in one of our executions. This is a rather strange result, since it is not common to obtain the maximum possible accuracy in any classification task. This could be indicating that there is a mistake in the interpretation of the results or that we got really lucky to find extremely good representations. With these last points and considering that, in the code of the original repository they report approximately 92% and we got the complete 100% accuracy, we summarize in Table 16 the best model obtained and we state that the experiments were successful.

| ResNet | batch_size | regularization_loss | label_top_1_accuracy | label_top_5_accuracy |
|---|---|---|---|---|
| 50 | 128 | 0.2747 | 1 | 1 |

**Table 16:** Conclusion of the experiments with BYOL.

# 11 CONCLUSIONS AND FURTHER WORK

In this document, it has been presented an overview of representation learning using contrastive methods. Representation learning is one of the most studied problems in the machine learning field in the last years, so we provided a resume of the first methods that were used, the problems that were found to these methods and how new frameworks emerged to overcome the problems that the first methods were facing.

# ACKNOWLEDGEMENTS

I would like to thank my tutor, Professor Nicolás Pérez de la Blanca, who has greatly guided me in this work, patiently answering my questions and doubts about the topic.

I would also like to thank my friends for supporting me not only reading this work and providing feedback, but also during the whole bachelor's degree. Specially, thank you to my girlfriend Isabel, who has been my solid ground and has pushed me to be constant and resilient.

Finally, I would like to thank my family, my very special, weird, but full of love family.

Thank you all.

Part V

# APPENDIX

# A APPENDIX A

This appendix will be used to set forth some theoretical results that might not always be relevant but are needed to understand some details during this thesis. Not all of them will be proven.

**Proposition 9** (Jensen's Inequality). *Let $f : \mathcal{D} \to \mathbb{R}$ be a concave function and $n \in \mathbb{N}$. For any $p_1, \ldots, p_n \in \mathbb{R}_0^+$ with $\sum p_i = 1$ and any $x_1, \ldots, x_n \in \mathcal{D}$, it holds that:*

$$\sum_{i=1}^{n} p_i f(x_i) \leq f\left(\sum_{i=1}^{n} p_i x_i\right).$$

*Furthermore, if $f$ is* strictly *concave and $p_i \geq 0$ for all $i = 1, \ldots, n$, then the equality holds if, and only if, $x_1 = \cdots = x_n$.*

In Chapter 6, the norm $\|\cdot\|_2$ is mentioned. Norm theory is a very extense field, so we will only mention the definition and the norm that we will use in the text.

**Definition A.1.** Given a vector space $X$ over a subfield $F$ of the complex numbers $\mathbb{C}$, a *norm* is a real valued function $\|\cdot\| : X \to \mathbb{R}$ with the following properties:

1. Triangle inequality, that is: $\|x + y\| \leq \|x\| + \|y\|$ for all $x, y \in X$.
2. Absolute homogeinity, that is: $\|sx\| = |s| \, \|x\|$ for all $x \in X$ and any scalar $s$.
3. Positive definiteness, that is $\|x\| \geq 0$ for all $x \in X$ and $\|x\| = 0$ if, and only if, $x = 0$.

In particular, the $\|\cdot\|_2$ that we used in the euclidean space $\mathbb{R}^n$, is defined as follows:

$$\|x\|_2 := \left(\sum_{i=1}^{n} x_i^2\right)^{\frac{1}{2}}, \qquad \forall x \in \mathbb{R}^n.$$

# B BIBLIOGRAPHY


Beery, Sara, van Horn, Grant, & Perona, Pietro. 2018. *Recognition in Terra Incognita*.

Belghazi, Mohamed Ishmael, Baratin, Aristide, Rajeswar, Sai, Ozair, Sherjil, Bengio, Yoshua, Courville, Aaron, & Hjelm, R Devon. 2018. *MINE: Mutual Information Neural Estimation*.

Bengio, Yoshua, Courville, Aaron, & Vincent, Pascal. 2014. Representation Learning: A Review and New Perspectives. *arXiv:1206.5538 [cs]*, Apr. arXiv: 1206.5538.

Caron, Mathilde, Bojanowski, Piotr, Joulin, Armand, & Douze, Matthijs. 2019. *Deep Clustering for Unsupervised Learning of Visual Features*.

Chen, Ting, Kornblith, Simon, Swersky, Kevin, Norouzi, Mohammad, & Hinton, Geoffrey. 2020a. *Big Self-Supervised Models are Strong Semi-Supervised Learners*.

Chen, Ting, Kornblith, Simon, Norouzi, Mohammad, & Hinton, Geoffrey. 2020b. A Simple Framework for Contrastive Learning of Visual Representations. *arXiv:2002.05709 [cs, stat]*, June. arXiv: 2002.05709.

Cosma, Rohilla Shalizi. 2021. *Advanced Data Analysis from an Elementary Point of View*.

Cover, T. M., & Thomas, Joy A. 1991. *Elements of information theory*. Wiley series in telecommunications. New York: Wiley.

Deng, Li. 2014. Deep Learning: Methods and Applications. *Foundations and Trends® in Signal Processing*, 7(3-4), 197–387.

Doersch, Carl, Gupta, Abhinav, & Efros, Alexei A. 2016. Unsupervised Visual Representation Learning by Context Prediction. *arXiv:1505.05192 [cs]*, Jan. arXiv: 1505.05192.

Grill, Jean-Bastien, Strub, Florian, Altché, Florent, Tallec, Corentin, Richemond, Pierre H., Buchatskaya, Elena, Doersch, Carl, Pires, Bernardo Avila, Guo, Zhaohan Daniel, Azar, Mohammad Gheshlaghi, Piot, Bilal, Kavukcuoglu, Koray, Munos, Rémi, & Valko, Michal. 2020. *Bootstrap your own latent: A new approach to self-supervised Learning*.

Gutmann, Michael U, & Hyvarinen, Aapo. Noise-Contrastive Estimation of Unnormalized Statistical Models, with Applications to Natural Image Statistics. 55.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, & Sun, Jian. 2015. *Deep Residual Learning for Image Recognition*.

He, Kaiming, Fan, Haoqi, Wu, Yuxin, Xie, Saining, & Girshick, Ross. 2020a. Momentum Contrast for Unsupervised Visual Representation Learning. *arXiv:1911.05722 [cs]*, Mar. arXiv: 1911.05722.

He, Kaiming, Fan, Haoqi, Wu, Yuxin, Xie, Saining, & Girshick, Ross. 2020b. *Momentum Contrast for Unsupervised Visual Representation Learning*.

Hjelm, R. Devon, Fedorov, Alex, Lavoie-Marchildon, Samuel, Grewal, Karan, Bachman, Phil, Trischler, Adam, & Bengio, Yoshua. 2019. Learning deep representations by mutual information estimation and maximization. *arXiv:1808.06670 [cs, stat]*, Feb. arXiv: 1808.06670.

Hénaff, Olivier J., Srinivas, Aravind, Fauw, Jeffrey De, Razavi, Ali, Doersch, Carl, Eslami, S. M. Ali, & van den Oord, Aaron. 2020. *Data-Efficient Image Recognition with Contrastive Predictive Coding*.

Krizhevsky, Alex. Learning Multiple Layers of Features from Tiny Images. 60.

Larochelle, Hugo, & Murray, Iain. The Neural Autoregressive Distribution Estimator. 9.

Löwe, Sindy, O'Connor, Peter, & Veeling, Bastiaan. 2019. Putting an End to End-to-End: Gradient-Isolated Learning of Representations. *Pages 3039–3051 of: Advances in Neural Information Processing Systems*.

Mikolov, Tomas, Chen, Kai, Corrado, Greg, & Dean, Jeffrey. 2013. Efficient Estimation of Word Representations in Vector Space. *arXiv:1301.3781 [cs]*, Sept. arXiv: 1301.3781.

Mitchell, Tom M. 1997. *Machine Learning*. McGraw-Hill series in computer science. New York: McGraw-Hill.

Oord, Aaron van den, Li, Yazhe, & Vinyals, Oriol. 2019. Representation Learning with Contrastive Predictive Coding. *arXiv:1807.03748 [cs, stat]*, Jan. arXiv: 1807.03748.

Poole, Ben, Ozair, Sherjil, Oord, Aaron van den, Alemi, Alexander A., & Tucker, George. 2019. On Variational Bounds of Mutual Information. *arXiv:1905.06922 [cs, stat]*, May. arXiv: 1905.06922.

Russakovsky, Olga, Deng, Jia, Su, Hao, Krause, Jonathan, Satheesh, Sanjeev, Ma, Sean, Huang, Zhiheng, Karpathy, Andrej, Khosla, Aditya, Bernstein, Michael, Berg, Alexander C., & Fei-Fei, Li. 2015. *ImageNet Large Scale Visual Recognition Challenge*.

Sohn, Kihyuk. 2016. Improved Deep Metric Learning with Multi-class N-pair Loss Objective. *In: NIPS*.

Tian, Yonglong, Sun, Chen, Poole, Ben, Krishnan, Dilip, Schmid, Cordelia, & Isola, Phillip. 2020. What Makes for Good Views for Contrastive Learning? *arXiv:2005.10243 [cs]*, Dec. arXiv: 2005.10243.

Tschannen, Michael, Djolonga, Josip, Rubenstein, Paul K., Gelly, Sylvain, & Lucic, Mario. 2020. On Mutual Information Maximization for Representation Learning. *arXiv:1907.13625 [cs, stat]*, Jan. arXiv: 1907.13625.

Uria, Benigno, Murray, Iain, & Larochelle, Hugo. 2014. RNADE: The real-valued neural autoregressive density-estimator. *arXiv:1306.0186 [cs, stat]*, Jan. arXiv: 1306.0186.

Wiskott, Laurenz, & Sejnowski, Terrence J. 2002. Slow Feature Analysis: Unsupervised Learning of Invariances. *Neural Computation*, **14**(4), 715–770.