Alma Mater Studiorum · University of Bologna

School of Science
Department of Physics and Astronomy
Master Degree in Physics

Estimation of biological vascular ageing via photoplethysmography: a comparison between statistical learning and deep learning

Supervisor:
Prof. Gastone Castellani

Submitted by:
Lorenzo Dall'Olio

Academic Year 2019/2020

Abstract

This work aims to exploit the biological ageing phenomena which affects human blood vessels. The analysis is performed starting from a database of photoplethysmographic signals acquired through smartphones. The further step involves a preprocessing phase, where the signals are detrended using a central moving average filter, demoduled using the envelope of the analytic signal obtained from the Hilbert transform, denoised using the central moving average filter over the envelope.

After the preprocessing we compared two different approaches. The first one regards Statistical Learning, which involves feature extraction and selection through the usage of statistics and machine learning algorithms. This in order to perform a classification supervised task over the chronological age of the individual, which is used as a proxy for healthy/non healthy vascular ageing.

The second one regards Deep Learning, which involves the realisation of a convolutional neural network to perform the same task, but avoiding the feature extraction/selection step and so possible bias introduced by such phases.

Doing so we obtained comparable outcomes in terms of area under the curve metrics from a 12 layers ResNet convolutional network and a support vector machine using just covariates together with a couple of extracted features, acquiring clues regarding the possible usage of such features as biomarkers for the vascular ageing process. The two mentioned features can be related with increasing arterial stiffness and increasing signal randomness due to ageing.

Contents

Chapter 1

Physiological Background: Photoplethysmography and Vascular Ageing

1.1 Photoplethysmography

The word plethysmography comes from the union of two ancient Greek words: 'plethysmos' (increase) and 'graph' (write) [1]. A plethysmograph measures the variation of volume of a given part of the body. It becomes intuitive the fact that a photo-plethysmograph will use light to accomplish its task.

In particular, the photoplethysmograph measures the variation of arteries volume over time, and this thanks to the variation of the amount of light that is diffused or absorbed by the tissues and therefore the variation in the amount of light transmitted/reflected. So, in the case of a reflectance mode photoplethysmograph, the amount of backscattered light will correspond to some variation variation of the blood [2].

To understand this mechanism we can start from the Lambert-Beer law applied to the model of a blood vessel:

$$A = \epsilon_\lambda l M \tag{1.1}$$

where ϵ_λ is the molar absorption coefficient for a given wavelength λ, l is the length of the optical path travelled by light, M is the molarity of the solution in which light travels and A is the resulting absorbance of light. Of course different tissues (such as bones, muscles, nerves, skin, arteries, etc.) will have different values for every of the r.h.s elements of eq. 1.1. The total absorbance will be given by the sum of the partial absorbance due to each single tissue. The most important consideration regarding photoplethysmography (PPG) is the fact that we do not measure absorbance by itself, but absorbance's variation over time. It seems reasonable to assume that ϵ_λ and M will be constant over time, once we fixed the tissue. Therefore the main variation will be given by l, which represents a different optical path length due to tissue's volume variation. Moreover, most of the tissues have a quite constant volume over time, such as bones or skin. The most appreciable variations will be due to blood vessels, as heart beat generates a blood pressure pulse that travels inside them, causing an increase in

volume. Since arteries walls are highly more elastic than veins walls, they will experience a much higher variation of volume as the blood pressure raises due to each heart beat. Considering what we just said, we end up having:

$$\frac{dA}{dt} \approx \frac{dl_{arteries}}{dt} \tag{1.2}$$

1.2 PPG detection

PPG measuring device are made of a light source and a photodetector. The light source shall emit light of a precise λ, which usually is in the near infrared range, but also visible light can be used. The most important factor is that other tissues, and mostly skin, have a small enough absorption coefficient for that given λ. In order to have good PPG acquisition we also need a highly vascolarized region with a thin layer of skin, such as the wrists, the forehead, the earlobes or the fingertips. Based on the device structure (and on the body part) we can measure transmitted light (with tissue between source and detector) or reflected light (with source and detector on the same side of the monitored tissue) to acquire PPG signals, each of these two modes having its own advantages.

1.3 Possible factors affecting PPG recordings

As long as the assumption 1.2 is not holding, we cannot rely on the recorded PPG signals.

This is the case when the sensor is affected by not negligible movements. Movements of the light source, of patient, or photodetector can all rapidly affect the length of optical path. Moreover, sensor's displacement from its original location changes the point where measurement is happening, with possible consequent variation of tissue (affecting ϵ_{lambda} and M) and/or thickness (affecting l).

Another important consideration is the pressure that the sensor is applying on the skin. Again, a not negligible variation of pressure can increase/reduce the amplitude of the recorded signal by varying the $\Delta l_{arteries}$ due to heart beats.

Apart from rapid variations, also a constant pressure can cause an improper PPG record, if the mentioned pressure would result in being too high. An excessive pressure (e.g. sensor too tight) can cause veins' pulses to become measurable too, resulting in a wrong variation of optical path $\Delta l_{tot} = \Delta l_{arteries} + \Delta l_{veins}$.

1.4 What is ageing?

Before even starting our analysis we need to define our goals, and since our goal regards vascular ageing we need to define ageing. Our common idea of ageing is usually referred to some sort of uniform and constant process, which "ruins" in some way most of the aspects and properties of biological systems.

Here we must create an important distinction:

- the ***chronological age*** is what we use to define the time past since birth;

- the ***biological age*** is the term we use to refer to the average health state of a biological system.

It is important to notice that this distinction generates a sort of duality in the word ageing. In fact we can think to it as the natural flow of time or as the amount of measurable effects that we accumulated over such time. Therefore, it is possible for the two ageing to differ, having a more or less healthier system than what we would expect given the chronological age. Since the first definition carries only a trivial, objective, and clear information, it has no big interests to dig deep into it over this work. Therefore, we will focus on the ***biological age***.

Last but not least, it is important to underline that, since the biological ageing is the amount of measurable effects on a given system, we can talk of biological ageing not just for a whole human being, but also for organ systems, organs or even tissues, as long as we can clearly define the biological system we are referring to.

It seems therefore reasonable to realize the fact that different parts of our body can have different biological ages, since the amount of measurable effects will never be uniformly distributed over all the different organs, tissues, and so on.

Now we will delve into the definition of biological ageing, focusing then on the main objective of our study: ***vascular ageing***.

1.5 Vascular ageing

Since PPG can properly measure some properties of arteries, such as their elasticity, we will now focus on the effects that ageing has on these organs. We mentioned that biological ageing is the sum of measurable effects due to age, we need some important effects regarding arteries.

The arteries are the efferent blood vessels with respect to the heart, which means that they carry blood from the heart towards the rest of the body. Large arteries are rich in elastin and collagen, while small muscular arteries are rich in vascular smooth muscle [3].

Then it appears quite clear that one of the most important properties of the arteries is walls' elasticity, which is reported to decrease with age [3], due to reduction of elastin content, increase in collagen content, and calcification, causing a higher arterial stiffness. This condition is exaggerated in some states such as hypertension and diabetes, increasing the risk of cardiovascular diseases [4]. With increasing stiffness, the vessels walls get thicker and the inner diameter smaller. This causes the cardiovascular system much more difficulty in moving the same amount of blood towards the arteries.

Moreover, some studies investigated the effects of long term smoking on arterial wall properties, obtaining contradictory results depending on the used methods. It is however known that smoking is a risk factor for atherosclerosis and therefore may contribute to arterial stiffness [5].

Chapter 2

Mathematical Background - Statistical Learning

2.1 Introduction to Machine Learning

Machine Learning (ML) can be broadly defined as computational methods using *experience* to improve performances or to make more accurate predictions [6]. Here "experience" shall be read as "previously available amount of information", in the sense that these computational methods try in different ways (and this "different" is what divides ML into sub-fields) to use information previously obtained from data to modify their outcomes. For this reason the first thing we have to underline is that outcomes' quality hugely depends on available data (and their quality).

As we mentioned, it is possible to divide ML into sub-fields, depending on the task, the group of used algorithms, the availability of desired outcomes, and much more. Whenever we use some sort of ground truth for our outcomes we are dealing with ***supervised learning***, otherwise we will have an ***unsupervised learning***.

In the first case we could have a label on our data, and our task could consists in assigning the correct label to each data sample, which is named a ***classification*** task. Some classification examples could be: recognizing hand-written digits, recognizing the presence of a given tissue in some medical images, differentiating a noisy electrical signal from a clear one, and so on.

Another supervised learning task could be the ***regression*** task, where we are interested in modelling a function able to map input into ground truth. Some examples could be: guessing the price of a dress given some of its properties (e.g. material, brand, year), predicting the age of a person given some information regarding his biomarkers, and so on. It appears that classification tasks are just a discretization of regression tasks, and for some reason it is, but the main difference resides in their purpose. During classification the key point is guessing the correct label, during regression we are more interested in correctly approximating the function. In the second case we can accept more frequently errors regarding the exact label as long as we get closer and closer to the mapping function. Moreover, there is an unbridgeable difference in certain cases. In regression tasks we can define a sort of "distance" from the ground truth, while in many of the classification tasks, this distance is quite impossible to define. Think about classifying a fruit, would it be worse to classify an apple as a peach or as an apricot?

Therefore we put under the classification task all those problems where we miss a clear way to compute the measure of "how wrong are we?", but keep in mind that we could always introduce it in a custom way if it can pursue our purpose.

Examples of unsupervised learning are represented by whenever we cannot use a ground truth (because it does not exist or it is too difficult/expensive to acquire). The main techniques here are ***clustering*** and ***dimensionality reduction***.

The clustering case consists in grouping the data by some similarity that they show among the available attributes.

The dimensionality reduction case is sometimes also called manifold learning. It consists of using mathematical and statistical steps in order to try to map a high number of dimension into a lower one, with as low information loss as possible. This is mostly used to reduce the number of attributes or to simply exploit data distribution with human eye, trying to plot data into a 2D/3D sub-spaces that are more as much representative of the whole attributes set as possible.

In certain scenarios, when we use a mixed approach, we can talk of ***semi-supervised learning***. This is the case when we sometimes use the available ground truth and sometimes not. This last case can be a good approach when there is a big difference between data availability and label availability. For example the labelling process can be very expensive or time-demanding, or on the other hand we can have many easily accessible databases of unlabelled data and just a few ones of labelled data. In both cases, semi-supervised learning tries to get out the best by combining the two approaches.

Another interesting case is the ***reinforcement learning***, which is usually treated as a different sub-field because of its particular optimization strategy. This last approach usually consists in creating many agents able to interact with an environment. Such agents will be rewarded based on their actions. After some iterations, only some of the most rewarded agents will be used to create new agents.

The optimization follows a trial-error procedure instead of following a sort of gradient descent. Therefore, in reinforcement learning, instead of a continuous improvement we observe outcomes to progress in a stepped trend, with flat periods alternated to noticeable improvements. This because the space of possible solutions is exploited in an almost random way, but every time we encounter a better solution we save it and never lose it.

The main example of this sector is represented by genetic algorithms.

2.2 Definition of Statistical Learning

One of the main targets of Machine Learning, as described so far, is to mimic some function which maps available inputs into desired outputs, may it be the main goal, as for regression tasks, or just an important consequence of the main goal, as for all other mentioned procedures.

Usually we define a fragmentation of the machine learning field based on the nature of the process we are using to mimic such function. We will call ***Statistical Learning*** (SL) a particular framework for machine learning where we use functional analysis and statistics to achieve that process in an explainable way. On the other hand, one of the properties which characterizes the Deep Learning field is the fact that operations which converts inputs into outputs are performed in a complex way, resulting in humans incapability of completely understand, track and explain the happening process, giving

to DL its peculiarity of being known as a "black box" approach.

SL is therefore preferable whenever we are able to design a workflow of feature acquisition/extraction and feature selection. In this way we can understand in a much clearer way *why* are the inputs mapped into the outputs. On the opposite, we will use DL if features/data are hard to select or manage, paying a price in terms of explainability of the mapping process and hence also in terms of generalizability of the found solution.

2.3 Spectral Embedding

We will now describe the Spectral Embedding (SE) algorithm from the Scikit-learn python library, since we will use this non trivial function in our further analysis for exploratory purposes.

First let us define that a mathematical embedding is an injective and "structure preserving" map $f : A \to B$, where the structure preserving meaning depends on the mathematical context. For simplicity, since in our further use cases it will hold that $A, B \subseteq \mathbb{R}^n$ we will identify f as an embedding if $f(A) \subseteq B$.

The algorithm has the aim to find a low dimensional representation of the data, by looking for a non-linear embedding.

To do so, the first thing the algorithm needs is a way to compute a distance. Usually the typical euclidean distance is employed by default, but any properly defined distance would be fine as well.

We will now analyze the ***Laplacian Eigenmaps*** algorithm which is performed to achieve SE outcomes.

2.3.1 Step 1: Building an Affinity matrix

The computed distance among data points is used to create an *affinity matrix*. An affinity matrix is defined as follows:

$$A_{ij} = \begin{cases} 1 & \text{if } j \text{ is a neighbour of } i \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

First of all, let us clarify that A has all 0s along the main diagonal, because the definition of neighbor never has the data point itself as a candidate.

In general, it can happen that j is a neighbour of i but the vice versa does not hold. In such cases we will talk of *directed* affinity matrix (since it is originated from a directed graph). Usually, if not differently specified, the matrix A will be symmetric (and it will be originated from an undirected graph, where j neighbour of i implies the vice versa).

Now we need to properly define *when* j is a neighbour of i. Most common options to determine such relation are the followings:

- ***k nearest neighbours***, where we simply define a parameter k and for every possible i we take as neighbours the k data points which are closer to it.
- ***thresholding***, where instead of fixing the number of neighbours we fix a threshold value t and we define i and j neighbours if $\text{dist}(x, y) = ||x - y|| \leq t$.

Please notice that the second option always gives a symmetric affinity matrix, while the first option is not originating a commutative definition of neighbours, therefore (especially for large dataset) it is almost always going to originate a non symmetric affinity matrix.

There is an exception for the second case, where instead of using a threshold value and assign only 1s and 0s we use a different definition of affinity matrix:

$$A_{ij} = g(x_i, x_j) = g(||x_i - x_j||) \tag{2.2}$$

Therefore our matrix is going to be symmetric, since the dependence from x_i and x_j is only through their norm, and with values different from 1s and 0s.

In this last case, it is common to prefer a function whose value decreases for "far" elements and increases for "close" elements. This because the algorithm is built to work in a way such that close elements (read neighbours) are assigned a higher value (read 1) than more distant elements (whose assigned value is 0) and we need a function able to mimic this trend.

The most common choice for the function g of equation 2.2 is a *radial basis function*, defined as

$$g(||x_i - x_j||) = e^{-\gamma||x_i - x_j||^2} \tag{2.3}$$

where γ is a variable parameter, whose default value on scikit-learn is the dimensionality (read number of features) of the dataset x.

2.3.2 Step 2: Build the Laplacian matrix

The next step of the algorithm consists in building the Laplacian matrix.

To do so we first need to define the *degree matrix* D

$$D_{ij} = \begin{cases} \sum A_i & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

where the matrix A is obtained from either equation 2.1 or equation 2.2. It is immediate to notice that D is a diagonal matrix by definition. We must underline that if A is not symmetric, taking the i-th row (A_i) or the j-th column($A^T{}_j$) will give different result. Therefore we will have to specify if we want to use the indegree or the outdegree and change the formula 2.4 accordingly (by using respectively $\sum A_k$ or $\sum A^T{}_k$).

Now, having both D and A, we can compute the laplacian matrix, given by

$$L = D - A \tag{2.5}$$

which can eventually be normalized through

$$L = D^{-\frac{1}{2}} L D^{-\frac{1}{2}} \tag{2.6}$$

In literature it is suggested to use the normalized version [7].

2.3.3 Step 3: Eigenvalue Decomposition

Now that we have the Laplacian matrix, we will perform a classical eigenvalue decomposition over it.

Therefore we will compute the eigenvalues of the matrix L and we will order them in an increasing order. Let us remind that the laplacian matrix is a positive semidefinite matrix [8], which means that all of its eigenvalues must be ≥ 0. So we have

$$0 \leq \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n \qquad \text{where } L\mathbf{v_i} = \lambda_i \mathbf{v_i} \tag{2.7}$$

with $\mathbf{v_i}$ as the eigenvector associated to the i-th eigenvalue.

We will discard the first eigenvector since λ_1 will be equal to 0 because of the connected giant component of the graph, and of course 0 can be a multiple eigenvalue if and only if we have multiple connected components [8].

In real context data, it is plausible for large datasets (after a proper outlier removal) and properly tuned algorithm parameters to have only 1 connected component. Then we will take the m successive eigenvectors $(\mathbf{v_2}, \ldots, \mathbf{v_{2+m}})$, and project our input data onto them, to finally achieve an m-dimensional plot which should maintain as much as possible the main mathematical structures present in the higher dimensional input space.

Chapter 3

Mathematical Background - Deep Learning

In this chapter we are going to discuss another sub-branch of ML, which is Deep Learning (DL). The reason it deserves a stand alone section resides in its peculiar difference from everything we already mentioned. The attribute *deep* comes from the way deep learning models "understand". In these models, concepts are built in a layered fashion: the model starts with few simple concepts (they can be features or whatever) and then combines these simpler concepts to build more complex ones. Iterating this procedure we end up with a hierarchy of concepts, and if we draw a graph of such hierarchy the graph is deep and layered, that is the reason why we call it deep learning [9].

3.1 Artificial Neural Networks

The main protagonist of DL are Artificial Neural Networks(NNs).

In NN our fundamental units are called *neurons*. A neuron can be thought as a human neuron (that is the reason for its name), it can receive multiple inputs, it can manage those inputs and send one or more outputs. Usually we dispose neurons in stacks called *layers* (even if in some topologies, the concept of layer is quite hard to apply, such as for Boltzmann machines [10]). Usually, Two neurons of the same layer are not directly connected each other, but a neuron is linked to every neurons of the previous layer and the successive layer (even these assumptions can vary due to topology [10]. Apart from the first and the last layers, which are respectively the input and the output of our NN, all the middle layers are called hidden layers, and they are the ones creating the depth and the complexity of the concepts we were talking about.

3.2 Convolutional Neural Networks

We will now focus on the main aspect of deep learning that was used for analysis: what is and how to build and train a Convolutional Neural Network (CNN).

The property that differentiates CNNs from NNs is the usage of the convolution operation.

We will now describe how convolution works in two dimensions, bearing in mind that the conversion for lower or higher dimensional spaces is quite straightforward.

Let's assume we have a matrix S of dimension $M \times N$ and a smaller matrix K of dimension $m \times n$ named *kernel*, with $m < M$ and $n < N$ and both m, n as odd positive integers. Then the convolution of the matrix S with the kernel K gives a new matrix O, whose dimension may change based on the selected boundary conditions. In general the elements of the matrix O are given by:

$$O_{x,y} = \sum_{i=1}^{m} \sum_{j=1}^{n} S_{x-\frac{(m-1)}{2}+i, y-\frac{(n-1)}{2}+j} \cdot K_{i,j} \tag{3.1}$$

This is not a perfectly accurate formula, but it helps us understanding what a convolution operation is.

The requirement of having both m and n as odd numbers is just to uniquely identify the indexes in the output matrix, but this is not a restriction, since any even sized kernel could be zero-padded in order obtain a higher odd sized kernel.

The reason why we did not mention a precise shape for the matrix O resides in the choice of the boundary conditions to use when $x - (m-1)/2 + i$ is below 0 or above M (same for y). This problem will be discussed in the *padding* section.

In a CNN the presence of multiple hidden layers can be summarized as a chain of consecutive convolutions with kernels that can be different also in size.

The values inside each kernel are the trainable parameters of the CNN, which means that during the training process the aim is to change these values in order to improve the outcomes.

In the following sections we are going to discuss the *hyperparameters*, those parameters whose different value generates a different CNN, and are therefore impossible to train. In fact the usual procedure is to tune these hyperparameters by training different CNNs for a long enough amount of iterations and choosing the values that are performing the best. Not all the below mentioned sections will be about actual hyperparameters, but since their effect on the model is equivalent we will mention them here anyway.

3.2.1 Kernel dimension

The kernel dimension is one of the main hyperparameters. With its dimension we are specifying also its shape, since it is not a must to have squared/cubic kernels in 2D/3D. Increasing the dimension of the kernel is going to affect the total complexity of the model, since we are going to increase the number of weights that we will have to train.

In common practice kernel is usually shaped as a square/cube, and the side is an odd positive integer. The reason behind this is to avoid ambiguities in equation 3.1, but as we previously said, any even sized kernel can be easily converted into an odd sized one thanks to a zero-padding.

Moreover, during the last decade we observed a decreasing trend in the kernel dimension. This fact can be explained by a comparison of two consecutive 3×3 convolutions with a single 5×5 convolution.

We call receptive field the number of inputs on which our central output value is depending from. Recalling equation 3.1, the receptive field is given by all the distinct possible combinations of i and j values, which are exactly equal to the kernel dimension $m \times n$. Of course, it is possible to define the receptive field after composed convolutions,

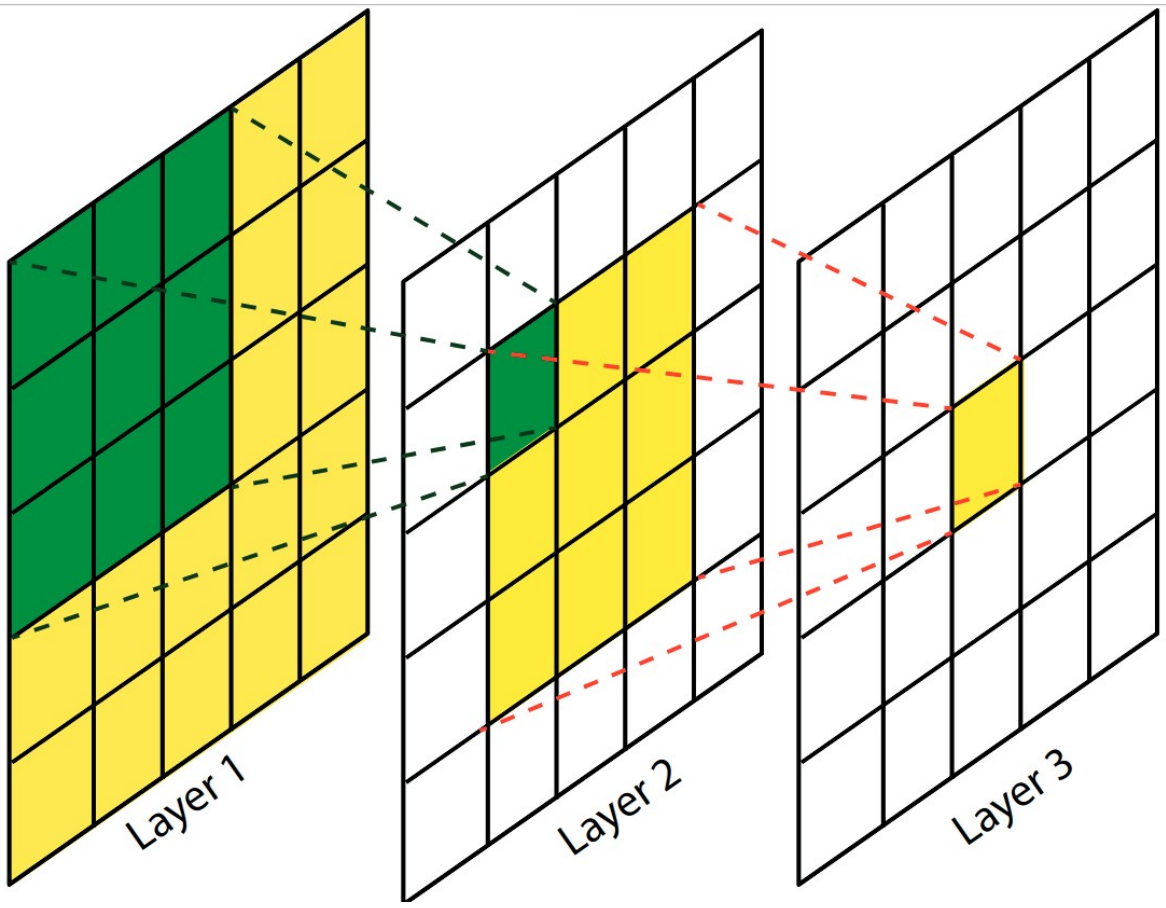

Figure 3.1: Comparison between two consecutive convolutions with kernel sizes of 3×3 (from layer 1 to layer 2 in green, and then from layer 2 to layer 3 in yellow), and a single convolution with kernel size of 5×5 (straight from layer 1 to layer 3 in yellow). Please notice that the central square of layer 3 has the same field of view in layer 1 (all the 25 squares) independently from which of the convolutions we consider.

with the consequence that we will have to keep into account overlaps. The reason why we are comparing these two cases is because they are two different way to obtain the same *receptive field*, as it can be seen in figure 3.1 [11].

Basically, we are comparing a *deeper* structure (more layers) with a *wider* structure (more parameters) The first case will need to train $2 \times (3 \times 3) = 18$ parameters, while the second one will need to train $5 \times 5 = 25$ parameters.

Therefore, even by having the same receptive field, the usage of two smaller kernels is preferable in terms of computational complexity. Moreover, the presence of two layers is going to "stratify" in a certain sense the effect of inputs, giving more importance to those ones which are closer to the center (and so detected and used by the first 3×3 convolution).

Apart from these two good effects in using deeper but narrower structures, there is also a negative downside. The iteration of two consecutive 3×3 convolutions cannot generate any possible 5×5 kernel, and this is quite logically reflected by the number of parameters previously compared.

Even given these considerations above, usually it should be preferable to use 3×3

kernels, since they are the smallest odd sized kernel which increases the receptive field. Anyway, it is common to see also 1×1 (used to change the number of channels/filters) and 5×5 kernels, while it is getting more and more rare to see kernels bigger than 7×7 among the state of the art performing networks.

3.2.2 Number of filters

The term *filter* is sometimes used as a synonym for kernel in images convolution. This hyperparameter is used to perform multiple convolutions in the same layer. The number of filters can also be thought as the number of channels in image related operations. For example, if we want to have a colored image as final output of a convolutional layer, we could need to fix its filters number to 3, which can be interpreted as the RGB channels in whichever order we may prefer.

More in general, we can think the number of filters as the number of neurons that the convolutional layer will have. Each neuron will perform a possibly different convolution changing its kernel's weights, and learning therefore a different feature that will be used from successive layers.

For this reason, whenever we are dealing with an n-dimensional kernel, the output of a convolutional layer will be (n+1)-dimensional due to the presence of the filters hyperparameter.

3.2.3 Strides and Padding

We will now consider *strides* and *padding* together, because they are both related to the "movement" of the kernel over the input.

The strides of an n-dimensional kernel is an n-dimensional sequence of integers, which specifies how many positions we need to move along a given direction before performing again the convolution operation. Recalling the definition we gave of convolution in equation 3.1, basically strides=1 means "perform the convolution operation for every possible (x,y) in the input". Strides=2 will mean "perform the convolution only once every two consecutive positions", so we are basically skipping half of the possible combinations and performing it only when (x,y) are both odd ((1,1), (1,3), (3,1), (3,3), ...).

We said that the strides has the same dimensionality of its kernel, because it is possible to specify different steps length for every dimension of the kernel. Therefore for a 2D convolution it can be possible to specify strides=(2,3), which means: "perform the convolution again only after moving of 2 positions on the first dimension or 3 positions on the second dimension".

The strides hyperparameter will affect the dimension and the shape of the output, the bigger the strides and the smaller the output.

The padding hyperparameter is the one we are using to deal with boundary conditions.

When we presented equation 3.1 we mentioned the presence of some unclear situations due to the fact that it can be required to access the matrix S in some position that does not exist (e.g. $S_{-1,-1}$). This kind of problem arise when a kernel, due to its dimension being bigger than 1, is centered on a boundary element of the input, and hence it escapes the input borders. For this reason we call it boundary conditions, and the most popular options are:

- *padding*=***valid***
- *padding*=***same***

The valid padding means that whenever the kernel exits from the input, that single operation must be skipped, resulting in an output of smaller dimension than the input (if the kernel is larger than 1 in size).

The same padding ideally stands for "keep the same dimension of the input". Even if this conclusion is not guaranteed (since other parameters can still reduce the dimension), the purpose of the same padding is to avoid skipping position because of ambiguity on how to treat the borders. Therefore the option padding=same is going to add, in any position that must be accessed but does not exist in the input, an imaginary position with the less interference possible in terms of its values. The value inside the imaginary position is going to depend on the operation, for convolutions it can insert zeros, for min or max pooling it will insert $\pm\inf$. Basically it will try to insert the most neutral value, just in order to not skip any of the convolution operations.

3.2.4 Activation functions

This section is not about a proper hyperparameter, but exactly as for a hyperparameter, the choice of the activation function is not trainable and must be tuned based on the purpose and the overall situation.

By activation function we define a function $f : \mathbb{R} \to \mathbb{R}$ which takes as input the result of a convolution for a single neuron and gives as output the final output of the neuron. This exact location allows these functions to regulate the output of a neuron and, more in general, its activation. From such behaviour this class of functions receives its name.

One of the main purposes of activation functions is to introduce non-linearity in the mapping process. In this way the learning process can model more complex features with fewer neurons and layers. We will now examine some of the most common choices for this particular class of functions, focusing on the most popular and interesting ones. Last but not least, we will focus also on activation functions' first derivative, since the training process requires its computation. Basically we are looking for the best compromise between non-linearity of the function and a computable and well defined first derivative.

Sigmoid function

The first activation function we are talking about is the sigmoid function, shown in figure 3.2. It is defined as:

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.2}$$

with derivative given by:

$$f'(x) = \sigma'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} \tag{3.3}$$

This function maps the input range from $]-\infty, +\infty[$ to $]0, 1[$ and it is non-linear by definition. One of its greatest advantages is having a continuous derivative, in fact, it can be easily seen that the sigmoid is a C^∞ class function.

On the other hand, one of its main drawbacks is related to its output range. In deep structures the presence of many consecutive hidden layers with sigmoid as activation function lowers the gradient's magnitude [12], causing the well known *gradient vanishing* problem.

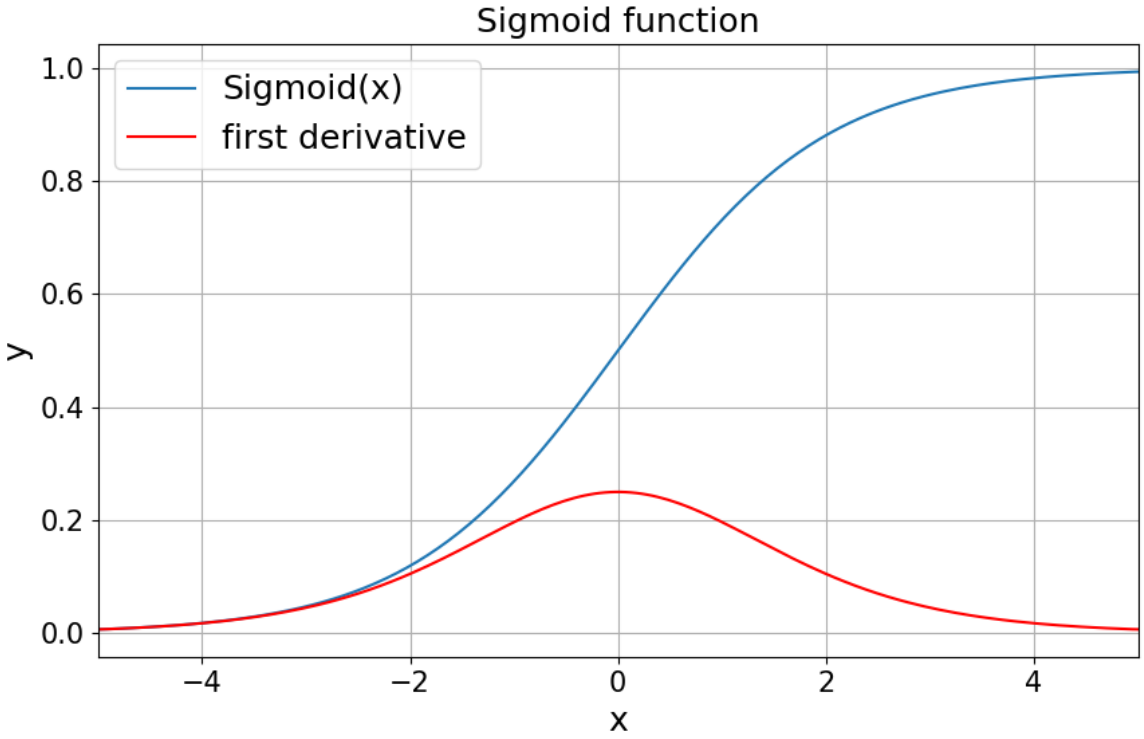

Figure 3.2: Sigmoid activation function and its first derivative, plotted for an input range from -5 to +5.

Hyperbolic tangent

Another C^∞ class activation function is given by the hyperbolic tangent, presented in figure 3.3. It is defined as:

$$f(x) = \tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.4}$$

with first derivative:

$$f'(x) = \tanh'(x) = \frac{4}{(e^x + e^{-x})^2} \tag{3.5}$$

It is important to notice that the hyperbolic tangent can be deduced from the sigmoid function, in fact:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{1 - e^{-2x}}{1 + e^{-2x}} = \frac{2}{1 + e^{-2x}} - 1 = 2\sigma(2x) - 1 \tag{3.6}$$

and

$$\tanh'(x) = \frac{4}{(e^x + e^{-x})^2} = \frac{4e^{-2x}}{(1 + e^{-2x})^2} = 2\sigma'(2x) - 1 \tag{3.7}$$

The existence of this relation gives to the hyperbolic tangent some common advantages and disadvantages of the sigmoid.

An important difference is the output range, which is now $]-1, +1[$. The fact that we have a codomain centered around the 0 gives a higher chance to have output values closer to 0. Moreover, its derivative is steeper than the sigmoid one. Therefore hyperbolic tangent is usually preferred to sigmoid because neural network tend to converge faster [13] and these networks tend to have a lower classification error [14].

Still, this function inherits the gradient vanishing problem from the sigmoid function.

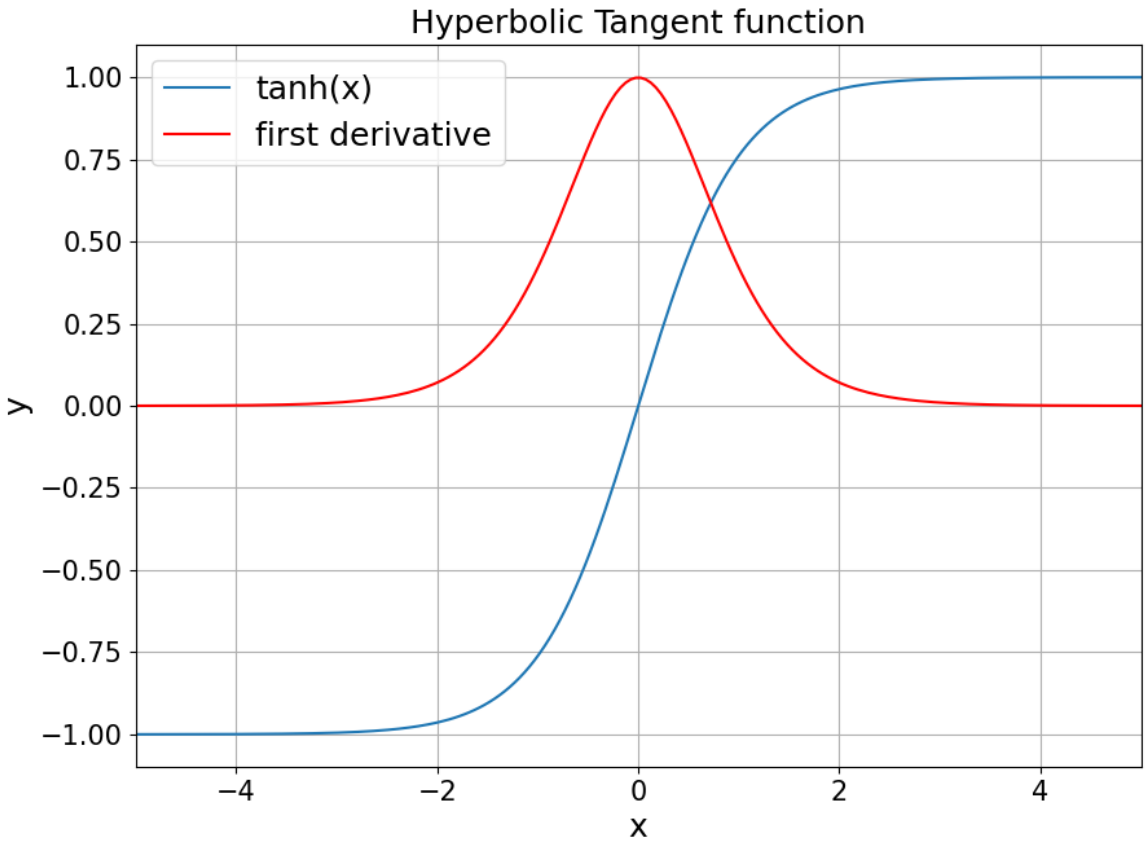

Figure 3.3: Hyperbolic tangent activation function and its first derivative, plotted for an input range from -5 to +5.

ReLU

The next function is the Rectifier Linear Unit (ReLu) reported in figure 3.4.

It is defined as:

$$f(x) = ReLU(x) = \max(0, x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \tag{3.8}$$

with obvious first derivative

$$f'(x) = ReLU'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases} \tag{3.9}$$

This is the most used activation function in the DL context [12].

One of its biggest advantages is being very simple, computationally speaking, and therefore faster to compute. This leads to a faster training and therefore a faster learning and better CNN performances [15].

Moreover, ReLU's derivative is constantly equal to 1 if the input is positive. This avoids the fact of gradient getting smaller and smaller among layers during backpropagation, and therefore it solves the gradient vanishing problem.

Some drawbacks are related to the fact that every non positive input is mapped into 0, avoiding the possibility to treat differently any of these values (this problem is also called "dying ReLU" [12]). If on one side, the possibility to output a true zero leads more easily to a sparse representation of data layer by layer (thanks to the presence of many zeros), the same effect can lead to the "death" of some neurons. This means that some neurons are never going to be updated and used if they end up in a weight configuration that gives them negative inputs, since the derivative of $ReLU(x)$ is 0 for $x \leq 0$ and then the backpropagation algorithm is going to have null effect on that given neuron. So once the neuron goes negative it is very unlikely for it to recover from this situation [16] [17].

Another big disadvantage is the fact that average output is identically positive, leading to a shift in bias for the next layer, process which slows down the training.

ELU

One of the proposal to try to improve ReLU is represented by the Exponential Linear Unit (ELU), reported in figure 3.5.

The function is defined as [18]:

$$f(x) = ELU(x) = \begin{cases} \alpha(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \tag{3.10}$$

with first derivative:

$$f'(x) = ELU'(x) = \begin{cases} \alpha e^x & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases} \tag{3.11}$$

ELU tries to solve the bias shift problem by moving its average closer to 0. This is possible since ELU menages to output negative values. This leads also to a faster training [19].

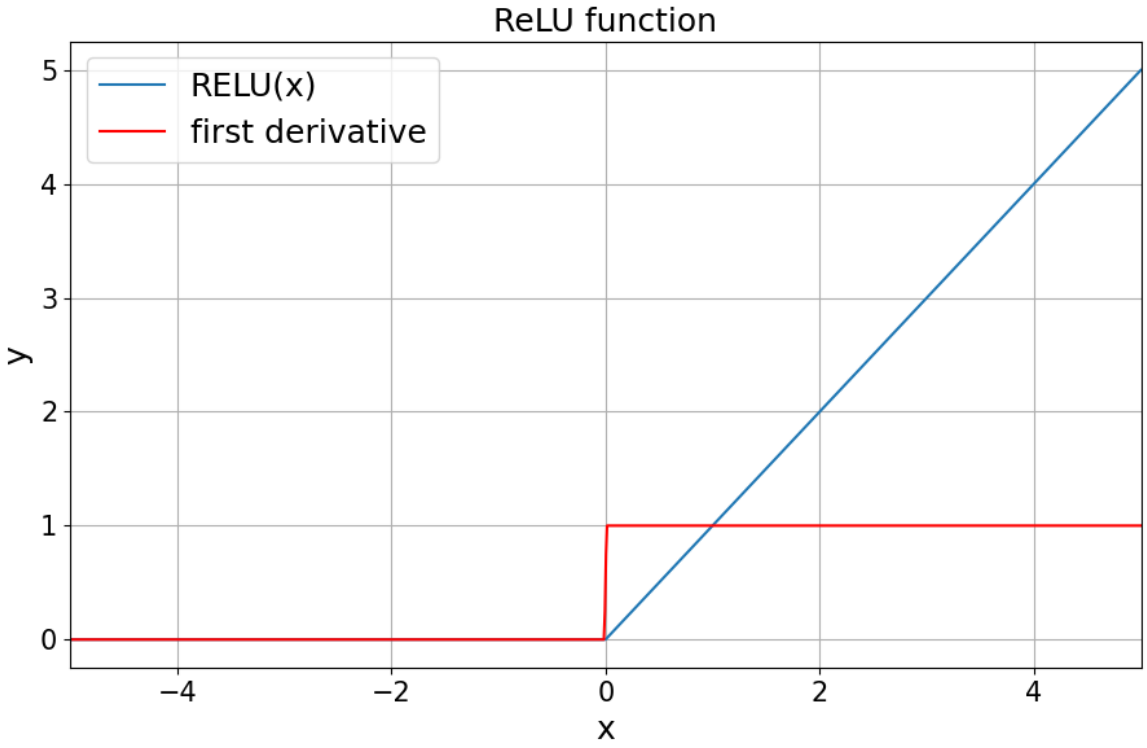

Figure 3.4: ReLU activation function and its first derivative, plotted for an input range from -5 to +5.

The α parameter is the one which controls the lower bounded saturation, and it is usually set equal to 1 for computational simplicity, but a fine tuning could be operated also on this parameter.

ELU succeeded in obtaining higher classification scores than ReLU [18], and in being more robust to input perturbation or noise [19].

SELU

The Scaled Exponential Linear Unit (SELU) is depicted in figure 3.6, and it is a modern variant of ELU.

It is defined as [20]:

$$f(x) = SELU(x) = \begin{cases} \gamma\alpha(e^x - 1) & \text{if } x \leq 0 \\ \gamma x & \text{if } x > 0 \end{cases} \tag{3.12}$$

with first derivative:

$$f'(x) = SELU'(x) = \begin{cases} \gamma\alpha e^x & \text{if } x \leq 0 \\ \gamma & \text{if } x > 0 \end{cases} \tag{3.13}$$

It appears clear that $SELU(x) = \gamma \cdot ELU(x)$, but the authors suggest also two very precise values for the parameters:

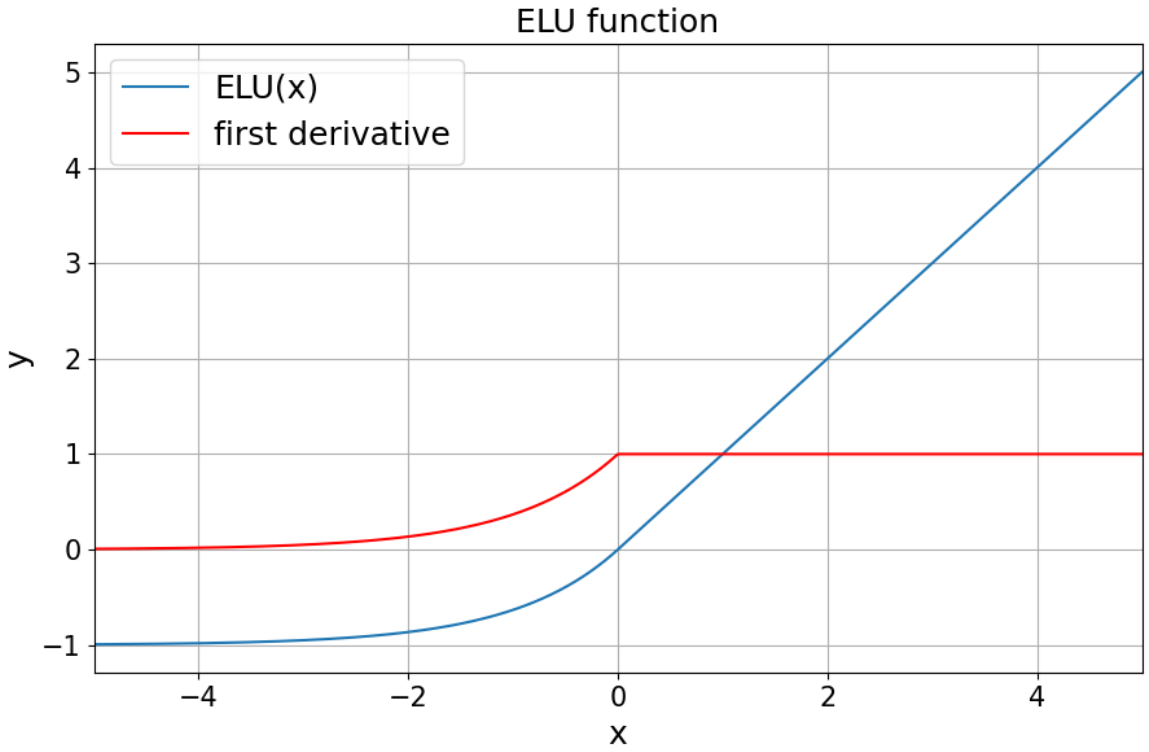

Figure 3.5: ELU activation function with $\alpha = 1.0$ and its first derivative, plotted for an input range from -5 to +5.

- $\alpha = 1.67326324$
- $\gamma = 1.05070098$

This because what SELU tries to implement is a way to self-normalize the weights, by creating a stable and attractive point in the weights space with 0 average and a variance of 1.

This kind of activation function was thought for feed forward NN, in order to allow deeper structures.

Even if SELU was successfully used in some cases with CNN [21], its general usage is still under study.

Softmax function

We will close the activation function discussion with the Softmax function.

Differently from previously discussed functions, Softmax is almost always used only for the last layer (output layer).

The reason resides in its definition:

$$f(x_i) = Softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{3.14}$$

where x_i is one of the n elements of the vector x.

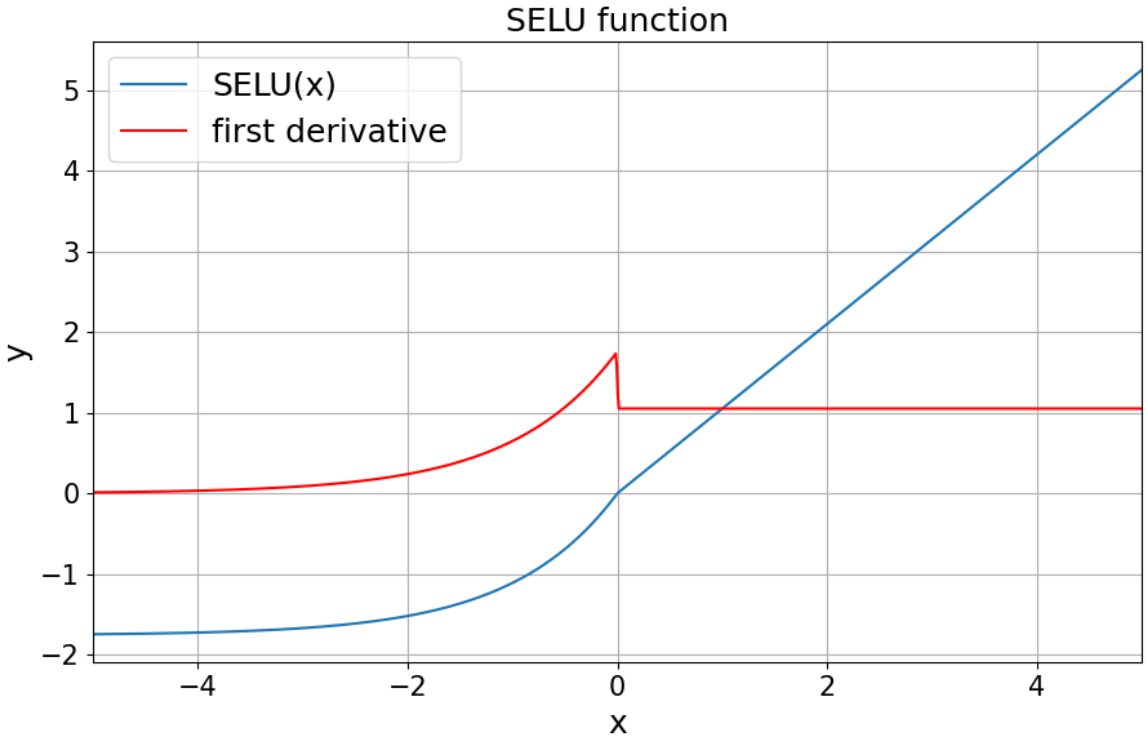

Figure 3.6: SELU activation function for $\alpha = 1.67326324$ and $\gamma = 1.05070098$, together with its first derivative, plotted for an input range from -5 to +5.

Therefore its main usage is as activation function for the output layer in a classification task NN, since in that case it basically returns the probability for a sample to belong to each class, usually with the one hot encoding among classes used as target.

3.2.5 Initializers

During this and the next section we will talk more in general of functions applied to any sort of trainable parameter, will it be a kernel weight or a bias (additive constant used to shift the input of the activation function). Then keep in mind that there will exist kernel initializers and bias initializers (and the same for the regularizers).

The used initializers can be thought as hyperparameters, but they are more precisely the initial condition of our path across the parameter space during the training process. Most of the initializers assign pseudo-random generated values to the parameters, leading therefore to different and possibly very distant outcomes achieved by the same initializer over different runs, if the random state is not fixed.

The first choice we have to perform is between:

- ***constant initializers***, where all parameters are chosen equal to a certain constant value;

- ***random initializers***, where all parameters are pseudo-randomly generated from a particular distribution;

- ***custom initializer***, where we are assigning a value to every parameter, or we define our own new way to assign them a value.

For the biases, the most common choice is the constant zero initializers, since we usually prefer to start with no bias and then introduce it if the training requires it.

For the kernel weights, the most common choice is among the random initializers. The most popular ones all made use of one between the uniform distribution and the gaussian distribution, and this choice is probably the most important of the whole initializers selection.

Once we have decided which distribution to use, we can take one further decision which regards the parameter of the distribution and its eventual truncation.

Therefore we can perform our last choice between the followings:

- ***Standard***, where the uniform distribution has a range of [0, 1] and the gaussian distribution has a standard deviation $\sigma = 1.0$.

- ***Glorot*** [14], where we introduce a dependence from the number of inputs f_{IN} and the number of outputs f_{OUT}, giving therefore a uniform distribution over $[-\sqrt{6/(f_{IN} + f_{OUT})}, \sqrt{6/(f_{IN} + f_{OUT})}]$ or a gaussian distribution with a standard deviation $\sigma = \sqrt{2/(f_{IN} + f_{OUT})}$.

- ***Lecun*** [13], where we only introduce the dependence from the number of inputs, and we will have a uniform distribution over $[-\sqrt{3/f_{IN}}, \sqrt{3/f_{IN}}]$ or a gaussian distribution with a standard deviation $\sigma = \sqrt{1/f_{IN}}$.

- ***He*** [15], which is exactly as the previous one times $\sqrt{2}$, therefore we will have a uniform distribution over $[-\sqrt{6/f_{IN}}, \sqrt{6/f_{IN}}]$ or a gaussian distribution with a standard deviation $\sigma = \sqrt{2/f_{IN}}$.

All the above mentioned gaussian distribution will clearly have mean $\mu = 0$.

3.2.6 Regularizers

As previously mentioned, we will talk about regularizers which will be equally applicable to kernel's weights or biases.

What regularizers try to do is avoiding weights/biases exploding values. To achieve this result, a penalization parameter can be applied to the norm of the weights/biases vectors.

The most common options are:

- ***No regularizer***.

- ***L1 regularizer***, where the penalization parameter multiplies the L1-norm of the mentioned vectors.

- ***L2 regularizer***, where the penalization parameter multiplies the L2-norm of the mentioned vectors.

- ***L1-L2 regularizer***, where we have two penalization parameters, one that multiplies the L1-norm, and one that multiplies the L2-norm of the mentioned vectors.

Usually the most common choice is to apply no regularization if no problems of exploding gradient are encountered, since every regularizer introduces some sort of bias in the values.

The L1 regularizer tends to shrink exactly to 0 all the groups of highly correlated values except one value per group, which by the way can vary with the run.

The L2 regularizer shrinks closer to 0 every element of a highly correlated group of values, without cancelling any value at all.

The L1-L2 regularizer is of course a tradeoff between these two effects, where the balancing is regulated by the ratio of the two penalization parameters.

3.2.7 Number of layers

This is one of the most important hyperparameters.

Its usage is quite intuitive, the more layers, the higher the number of parameters in the network and hence the bigger the complexity of the CNN.

The usual trade-off is between underfitting (due to a low complexity structure) and overfitting (caused by a high complexity CNN approaching a simpler problem).

Even fixing the main parameter regarding structure complexity, which is the total number of neurons, there is another trade off that we have to find. This second one is between "width" and "depth".

Let us consider, for simplicity's sake, total number of neurons in a CNN as a synonym for CNN's complexity. Fixing the total number of neurons at N implies that we still have many possible architectures to choose from.

To count them we need to find how many different ways do exist to sum natural numbers up to the value N. In fact the total number of neurons in a CNN is obtained by summing the number of neurons for each layer

$$N = \sum_{i=1}^{l} n_i \tag{3.15}$$

where l is the number of layers in the CNN.

For our counting purpose we will use an ordered tuple as notation:

$$(a, b, c, \ldots) = a + b + c + \ldots \tag{3.16}$$

The reason for the use of an ordered tuple stands in the fact that, for our purpose (2,1) is different from (1,2), based on the fact that we will have different CNN structures, respectively of decreasing or increasing width.

Now, to find how many different ordered tuples sum up to N we proceed by thinking N as a sequence of N '+1' adding factors:

$$N = (+1 + 1 + 1 \ldots + 1 + 1) \tag{3.17}$$

now we notice that, between any two of these factors, we can replace the '+' sign with a sign surrounded by parenthesis ')+(', thus creating a new ordered tuple.

$$(+1+1)+(1+1\ldots+1)=(2,N-2)=N \tag{3.18}$$

Therefore, if we have N '+1' elements there are exactly $N-1$ different locations where we can independently replace '+' with ')+(', and any of these replacements generates a new ordered tuple.

Considering that for $N-1$ position we either have a '+' or a ')+(', the number of different ordered tuples that sum up to N is given by $2^{(N-1)}$, and that is also the number of different architectures of a CNN that have the same total number of neurons but a different structure.

It appears quite clear that, solving the optimization problem by brute force (which means trying all the different $2^{(N-1)}$ CNNs, and choose the best one) is prohibitive already for small numbers such as $N=10$.

Usually, fixed N, it is preferable to go deeper than wider. The reason resides in the definition of Deep Learning, since the features or concepts that the CNN will learn are built in a deep way. Therefore using a single layer with N neurons is going to create N simple features, while using 2 layers with $N/2$ neurons will create $N/2$ more complex features combining $N/2$ simpler features.

For these reasons, the approach is usually to tune most hyperparameters on a couple of layers, and then increase the number of layers paying attention to overfitting and reducing the number of neurons per layer if necessary. Of course the number of neurons per layer is not constant and it is possible to change width along the layers' depth. The most common structures are of constant or decreasing number of neurons as the layer's depth increases, but it is possible to encounter also the opposite.

3.2.8 Kind of layers

This is the first example of something that is not a hyperparameter but has the same effect on the model. In fact, it s obviously not possible to change layer type during the training process, and so it must be fixed each time. But why are we talking of layer "kinds"? Because it is possible to have layers performing different operation from the convolution.

Now we will exploit some of these other possibilities, excluding some very peculiar kinds (such as long-short term memory cells, used almost only in recurrent NN) and focusing on the mostly used options.

Pooling layers

In CNNs the most common option is to alternate convolutional layers with pooling layers. A pooling operation consists in selecting a portion of the input and discard some information from it.

The reason why pooling layers are used is because we may want to reduce the amount of information stored in successive layers, together with CNN's complexity.

Most common cases for pooling operation consist of selecting a N-dimensional rectangular portion (which in case of global pooling is equivalent to the whole input) of an N-dimensional input, and then giving as an output a single number, which can be the maximum, the minimum, the average or other.

Usually pooling is used to get rid of some superficial information, then its usage must be accurately studied and placed.

Dropout

This one is not properly a layer kind, but it does behave like one.

The effect of a Dropout is taking as input previous layer neurons' outputs, and de-activate some of them by simply multiplying those for 0.

The reason why this should be used is restricted to the training phase, where de-activating some neurons outputs forces the neurons of the successive layers to do without some of their possible inputs. Of course the inputs to de-activate are chosen randomly among all the inputs, and they are possibly different for each iteration. This procedures generates a training less prone to overfitting and more robust to outliers.

The only parameter of a dropout layer is the fraction of input units to drop r. There are different versions of dropout, among which we will mention only α-dropout. α-dropout is a dropout which tries to maintain the mean and the variance input as much unvaried as possible even after the dropout operation. This last version is better suited for enhancing the SELU effect of self-normalizing NN [20].

Of course, dropout effects must be removed during the testing phase, otherwise they will act as a pruning operation on the CNN.

Input managing layers

Another different class of layers consists of those ones which simply manage the input.

There are many possibilities such as reshaping, scaling and/or translating the values, apply transformations.

In a certain sense, activation function can be seen as input managing layers since they simply apply a transformation to their inputs.

The most used layers of this kind which are not activation functions are:

- **Flatten**, which simply converts the input into a 1-dimensional sequence. Generally used as a bridge between a CNN's last layer and the first layer of a dense NN.
- **Batch Normalization**, which standardizes its input based on the minibatch samples, giving an output with the same shape, but in a way where the minibatch samples have mean close to 0 and standard deviation close to 1.

In particular the last one is used to "adjust" every minibatch. We will talk more deeply of minibatch later, but right now you can imagine that with minibatch we indicate the amount of different data samples used before updating the CNN parameters.

Hence, Batch Normalization is used to "regularize" the behaviour of the network in order to avoid a drift in the updates due to a small group of some systematically different input data.

3.2.9 Optimizer

The optimizer is literally the algorithm that we are going to use in order to update our CNN parameters during training. Moreover, this variable does not regard our CNN,

but it regards the training process. Therefore this is again, something which is not a hyperparameter but behaves like one.

All the optimizers will have possibly very different instructions and computations, leading to a different set of tunable parameters. The only parameter that every optimizer will have for sure is the *learning rate*. This particular parameter affects the magnitude of the updates to the CNN trainable parameters. Basically, after running the CNN over the training data, we will have some sort of "error" (which will be discussed later in the loss function section) the higher the learning rate the bigger the magnitude of the updates. We can think of it, as the training without an optimizer gives use the direction (in the trainable parameter space) for the best update, then the optimizer tells us how far to move along that given direction. The length of this step is going to depend in some way on the chosen learning rate. This does not always happen exactly, some optimizers use a mathematical tool called *momentum* which also affects the mentioned direction, but it gives us the idea of what the learning rate is.

Considering $\mathbf{p}$ as the vector of the CNN's trainable parameters, X as the data, y as the targets, and $f(\mathbf{p}; X, y)$ as the chosen loss function, we will now analyze some of the most popular optimizers and briefly explain their inner tunable parameters.

Gradient Descent and Stochastic Gradient Descent

Gradient Descent is one of the oldest optimization algorithms still used today (but mostly through its newer stochastic version).

What this algorithm does is using the slope of a loss function $f(\mathbf{p}; X, y)$ to update the vector of trainable parameters in the following way:

$$p_{i_{n+1}} = p_{i_n} - l\frac{d}{dp_i} f(\mathbf{p_n}; X, y) \tag{3.19}$$

where l is the learning rate, and p_{i_k} refers to the value of the parameter p_i for the k-th iteration.

The formula can obviously be rewritten as follows

$$\mathbf{p_{n+1}} = \mathbf{p_n} - l\nabla_{\mathbf{p}} f(\mathbf{p_n}; X, y) \tag{3.20}$$

where $\nabla_{\mathbf{p}}(f(\mathbf{p_n}))$ is the gradient of f with respect to the vector $\mathbf{p_n}$.

It is clear that, in order to be effective, we need a function f with a clear global minimum, even better we could require f to be convex. Such case is by the way not very realistic, most loss functions are not convex and usually they do have many local minima, where the training can get stuck if l is not big enough. Anyway, a learning rate too big could stop the training from converging (condition where the update is small enough to stop the training process), giving the updates an oscillatory trend. It is known that Gradient Descent can easily get stuck in poor local minima [22].

To tackle this problem, one solution is usually to start with a relatively big learning rate and apply a decay rate, so that we have a decreasing learning rate along with the number of iterations.

Another way to solve the local minima problem is the use of *momentum*.

Momentum technique consists in a technique for accelerating gradient descent by accumulating a velocity vector over a direction of persistent descent [23].

The new updates with the introduction of momentum can be written as

$$\mathbf{v_{n+1}} = \mu \mathbf{v_n} - l\nabla_{\mathbf{p}}(f(\mathbf{p_n}; X, y)) \tag{3.21}$$

$$\mathbf{p_{n+1}} = \mathbf{p_n} + \mathbf{v_{n+1}} \tag{3.22}$$

where μ is the value of the momentum and $\mathbf{v_k}$ is the vector of accumulated velocity at k-th iteration.

One interesting variant for the momentum is the Nesterov momentum.

This version has a different way to compute the velocity vector, trying in a certain sense to "guess" where the updates are going move the vector $\mathbf{p_{n+1}}$ and using this guess in the computation of the gradient. This results in the following updates formulae

$$\mathbf{v_{n+1}} = \mu \mathbf{v_n} - l\nabla_{\mathbf{p}}(f(\mathbf{p_n} + \mu \mathbf{v_n}; X, y)) \tag{3.23}$$

$$\mathbf{p_{n+1}} = \mathbf{p_n} + \mathbf{v_{n+1}} \tag{3.24}$$

The result of Nesterov momentum is an optimizer more able to decelerate when needed in certain situations [23].

The Stochastic Gradient Descent (SGD) is basically the same algorithm, but it randomly chooses a small subset of (X,y) and performs the updates just based on this subset.

SGD is a good way to speed up the optimization process, and it is particularly useful when there is some sort of redundancy in the training data. For example if we have clear clusters in X, then it is believable that data points from the same cluster will have similar effect over the gradient computation.

Anyway, the same effect of SGD can be obtained varying the batch dimension (whose effects will be discussed in a further session) and therefore it is common to use this second hyperparameter to turn GD into a SGD.

ADAM

One of the most recent and yet vastly used optimizer is the Adaptive Moment Estimation (ADAM) optimizer.

Firstly formulated by Diederik P. Kingma and Jimmy Lei Ba in 2015 the method is reported to have little memory requirements, be computationally efficient and well suited for problems that are large in terms of data and/or parameters [24].

The updates are more complex and with more intermediate steps with respect to SGD.

$$\mathbf{m_0} = \mathbf{v_0} = \mathbf{0} \tag{3.25}$$
$$\mathbf{m_{n+1}} = \beta_1 \mathbf{m_n} + (1-\beta_1)\nabla_{\mathbf{p}}(f(\mathbf{p_n}; X, y)) \tag{3.26}$$
$$\mathbf{v_{n+1}} = \beta_2 \mathbf{v_n} + (1-\beta_2)(\nabla_{\mathbf{p}}(f(\mathbf{p_n}; X, y)))^2 \tag{3.27}$$
$$\hat{\mathbf{m}}_{\mathbf{n+1}} = \frac{\mathbf{m_{n+1}}}{(1-\beta_1^{n+1})} \tag{3.28}$$
$$\hat{\mathbf{v}}_{\mathbf{n+1}} = \frac{\mathbf{v_{n+1}}}{(1-\beta_2^{n+1})} \tag{3.29}$$
$$\mathbf{p_{n+1}} = \mathbf{p_n} - l\frac{\hat{\mathbf{m}}_{\mathbf{n+1}}}{\sqrt{\hat{\mathbf{v}}_{\mathbf{n+1}}} + \epsilon} \tag{3.30}$$

where $(\mathbf{m}, \mathbf{v})$ are the first moment and the second raw moment vectors, with respectively their decay rates (β_1, β_2). What these vectors try to estimate are the first moment (the mean) and the second raw moment (the un-centered variance) of the gradient function, through an exponential moving average over past iterations. Moreover, since we want (β_1, β_2) to act as decay rates, we have $\beta_1, \beta_2 \in [0, 1[$ The version without hat of the moment vectors correspond to the biased estimate, since the vectors are initialized as full of zeros we will have a bias towards 0. The hat version indicates the un-biased estimate. Finally, ϵ is used to avoid divisions by 0 (which otherwise would always occur in the first iteration), and since it has no mathematical meaning it is advisable to use a very small value for it.

In literature there are some suggested values which are highly recommended [24]:

$$l = 0.001 \tag{3.31}$$
$$\beta_1 = 0.9 \tag{3.32}$$
$$\beta_2 = 0.999 \tag{3.33}$$
$$\epsilon = 10^{-8} \tag{3.34}$$

but it is possible to look for a better tuning of them, with particular reference to the learning rate.

ADAM has shown many advantages in terms of generalization properties, and it was found to be robust and well-suited to a wide range of non-convex optimization problems in the field machine learning [24].

On the other hand, some more results showed that ADAM fails to converge on some particular simple one-dimensional convex problems [25], proving once again that there is not a universal optimizer for the ML/DL tasks.

NADAM

The Nesterov Accelerated version of ADAM (NADAM) tries to improve even further the ADAM results.

The operation of "guessing" the next update and compute the gradient over that guessing is by the way much more complicated than the one we saw earlier in 3.23.

This operation is obtained by introducing a decay schedule for a new parameter μ which depends on β_1. We report the updates implemented by the NADAM algorithm (for precise explanations, see the Methods section of [Dozat, 2015] [26])

$$\mathbf{m_0} = \mathbf{v_0} = \mathbf{0} \tag{3.35}$$

$$\mu_n = \beta_1(1 - 0.5 \cdot 0.96^{(n)/250}) \tag{3.36}$$

$$\mathbf{m_n} = \beta_1 \mathbf{m_{n-1}} + (1 - \beta_1)\nabla_{\mathbf{p}}(f(\mathbf{p_{n-1}}; X, y)) \tag{3.37}$$

$$\mathbf{v_n} = \beta_2 \mathbf{v_{n-1}} + (1 - \beta_2)(\nabla_{\mathbf{p}}(f(\mathbf{p_{n-1}}; X, y)))^2 \tag{3.38}$$

$$\hat{\mathbf{m}}_\mathbf{n} = \frac{\mathbf{m_n}}{(1 - \prod_{i=1}^{1=n} \mu_i)} \tag{3.39}$$

$$\hat{\mathbf{v}}_\mathbf{n} = \frac{\mathbf{v_n}}{(1 - \beta_2^n)} \tag{3.40}$$

$$\bar{\mathbf{m}}_\mathbf{n} = \frac{(1 - \mu_n)\nabla_{\mathbf{p}}(f(\mathbf{p_{n-1}}; X, y))}{(1 - \prod_{i=1}^{1=n} \mu_i)} + \mu_{n+1}\hat{\mathbf{m}}_\mathbf{n} \tag{3.41}$$

$$\mathbf{p_n} = \mathbf{p_{n-1}} - l\frac{\bar{\mathbf{m}}_\mathbf{n}}{\sqrt{\hat{\mathbf{v}}_\mathbf{n}} + \epsilon} \tag{3.42}$$

The implementation of Nesterov version of momentum can be recognized in the step 3.41 for the computation of $\bar{\mathbf{m}}_n$, where now the "guess" is no more performed in the gradient computation, but in adding the "next" momentum $\mu_{n+1}\mathbf{m_n}$.

The author of NADAM suggest a slightly different value for one parameter, which is $\beta_1 = 0.99$.

From the comparison of NADAM and ADAM performances, it is common now to believe that the application Nesterov momentum leads in general to better results than the plain classic momentum.

3.2.10 Loss function

When we are dealing with CNN training we need a way to express how good are our results after each iteration.

In supervised learning we have some true targets and we want to reply those targets through our CNN. Therefore we will have some data samples X, with a true target vector $\mathbf{y_t}$ and, by running our CNN over the data samples, we will generate a vector of predicted targets $\mathbf{y_p}$ and compare it with the true ones.

Such comparison is evaluated in terms of a function $G(\mathbf{y_p}, \mathbf{y_t})$ which is called loss function or cost function.

Usually, the main requirement for $G(\mathbf{y_p}, \mathbf{y_t})$ is to have its global minimum when $\mathbf{y_p} = \mathbf{y_t}$.

Anyway, some other properties can be quite useful, such as being smooth and/or convex (having only 1 minimum and a non negative second derivative always).

We can distinct two different sub-classes of loss function, depending on the nature of the target vectors.

The ***Probabilistic loss functions*** are used in order to compare two vectors $\mathbf{y_p}$ and $\mathbf{y_t}$ which can be interpreted as probability distribution. The main use case for this functions are the *classification* tasks. In this cases usually the vector $\mathbf{y_t}$ is a vector of one hot encoded vectors, meaning that

$$\mathbf{y_{i,t}} = (0, 0, \ldots, 0, 1, 0, \ldots, 0, 0) \tag{3.43}$$

where $\mathbf{y_{i,t}}$ is the one hot encoded vector for the i-th sample, and its k-th element is 1 if the sample belongs to the k-th class, 0 otherwise. Then the vector $\mathbf{y_t}$ can be thought as a matrix, with one row for each sample and one column for each class, and the element $\mathbf{y_{ik}}$ is 1 if the i-th sample belongs to the k-th class, 0 otherwise. Therefore each row of $\mathbf{y_t}$ can be thought as a discrete normalized probability distribution.

On the other hand, the rows of $\mathbf{y_t}$ will be compared with the rows of $\mathbf{y_p}$ produced by our CNN. These rows are not going to be one hot encoded (even if we aim for it), since the values in the output layer of our CNN will rarely be exact integers (unless we force them to be with a step activation function).

We can overcome the problem of not having a one hot encoded output in terms of classification, by simply assigning the sample to the class j if the j-th neuron in the output layer is the one containing the maximum value all over that layer.

But, apart from guessing the correct class, we should do more in terms of comparison, and so we can interpret also the output layer as a discrete probability distribution (which will be our $\mathbf{y_{i,p}}$) and compare it with the true target probability distribution ($\mathbf{y_{i,t}}$).

In general, probabilistic loss function tend to treat differently some small variations that can occur in the predicted probability distribution, their main aim is to align the two distributions peaks.

On the other hand, the ***Regression loss functions*** fits more regression tasks, where even the smallest difference between $\mathbf{y_p}$ and $\mathbf{y_t}$ could be useful given the more continuous nature of the target, and therefore it will be considered.

By the way, the main difference between these two sub-classes is that the probabilistic loss functions are expected to work with probability distributions, therefore part of their efficiency is linked to the fact that the true targets and the CNN outputs are normalized.

In the followings, we will examine some of the most common loss functions and their expressions assuming we have the true targets $\mathbf{y_t}$ and the predicted target $\mathbf{y_p}$.

Categorical Crossentropy

This probabilistic loss function is by far the most used in classification tasks.

It is called crossentropy because it computes the entropy using the well-known Shannon's formula, but using two different probability distributions. Then its expression is the following:

$$\mathrm{CC}(\mathbf{y_t}, \mathbf{y_p}) = -\frac{\sum_{i=1}^{N} \sum_{j=1}^{d} \mathbf{y_{ij,t}} \cdot \ln(\mathbf{y_{ij,p}})}{N} \tag{3.44}$$

where d is the dimensionality of each single data sample. Sometimes, since N is the constant indicating the number of samples, we can multiply by N to move from the *average* categorical crossentropy to the *total* categorical crossentropy.

Please notice that, if all the $\mathbf{y_{i,t}}$ are one hot-encoded vectors, the crossentropy expression can be rewritten in a more compact way

$$\mathrm{CC}(\mathbf{y_t}, \mathbf{y_p}) = -\frac{\sum_{i=1}^{N} \ln(\mathbf{y_{ij,p}})}{N} \quad \text{with } j = \mathrm{argmax}(\mathbf{y_{i,t}}) \tag{3.45}$$

and this is clearly a probabilistic loss function since

$$\text{CC}([0,1,0],[0.0,0.7,0.2]) = \text{CC}([0,1,0],[0.15,0.7,0.15]) \tag{3.46}$$

therefore ignoring the different composition of $\mathbf{y_p}$ as long as its element in position argmax($\mathbf{y_t}$) is unchanged.

Poisson

A probabilistic loss function based on a similar criteria as the crossentropy one.

Its expression is

$$\text{Poisson}(\mathbf{y_t},\mathbf{y_p}) = -\frac{\sum_{i=1}^{N}\sum_{j=1}^{d}\mathbf{y_{ij,p}} - \mathbf{y_{ij,t}}\cdot\ln(\mathbf{y_{ij,p}})}{N} \tag{3.47}$$

This loss function is advisable when our samples' targets could follow a Poisson distribution. Such distribution is typical for the number of times an event happen on a specified time interval, assuming:

- events occur independently,
- The average rate at which events occur is independent of any occurrences,
- two events can not occur simultaneously,
- the number of occurrences k is a non-negative integer.

In that case the distribution is given by

$$P(k;\lambda) = P(x=k) = \frac{\lambda^k e^{-\lambda}}{k!} \tag{3.48}$$

where λ is a parameter proper of the distribution.

Kullback-Leibler Divergence

This probabilistic loss function, (usually mentioned as KLD) is expressed as

$$\text{KLD}(\mathbf{y_t},\mathbf{y_p}) = \frac{\sum_{i=1}^{N}\sum_{j=1}^{d}\mathbf{y_{ij,t}}\cdot\ln(\frac{\mathbf{y_{ij,t}}}{\mathbf{y_{ij,p}}})}{N} \tag{3.49}$$

Please notice that this expression is tightly connected with the categorical crossentropy one

$$\begin{aligned}\text{KLD}(\mathbf{y_t},\mathbf{y_p}) &= \frac{\sum_{i=1}^{N}\sum_{j=1}^{d}\mathbf{y_{ij,t}}\cdot\ln(\frac{\mathbf{y_{ij,t}}}{\mathbf{y_{ij,p}}})}{N}\\ &= \frac{\sum_{i=1}^{N}\sum_{j=1}^{d}\mathbf{y_{ij,t}}\cdot\ln(\mathbf{y_{ij,t}}) - \mathbf{y_{ij,t}}\cdot\ln(\mathbf{y_{ij,p}})}{N}\\ &= -\frac{H(\mathbf{y_t})}{N} - \frac{CC(\mathbf{y_t},\mathbf{y_p})}{N}\end{aligned} \tag{3.50}$$

and since the average entropy of the true targets is a constant, computing the Kullback-Leibler Divergence is completely identical to computing the categorical crossentropy with a bias.

Mean Squared Error

This loss function is a regression loss function, and by far the most used in this field.

It is defined as:

$$\mathrm{MSE}(\mathbf{y_t}, \mathbf{y_p}) = \frac{\sum_{i=1}^{N} \sum_{j=1}^{d} |\mathbf{y_{ij,t}} - \mathbf{y_{ij,p}}|^2}{N \cdot d} \tag{3.51}$$

where we obtain the *average MSE*. For this (and most of the further regression loss) function it is also possible to multiply by d to obtain the *average total MSE per sample*

This particular loss function is often used in regression problems because it is convex and it "punishes" the most when outputs are far from the target. This aspect implies also a known drawback of MSE: its strong dependence on outliers. It can be easily seen that, a huge difference for a single (i, j) value in formula 3.51 can be orders of magnitude over all the other differences, resulting in $\mathrm{MSE}(\mathbf{y_t}, \mathbf{y_p}) \approx \mathrm{MSE_{ij}}(\mathbf{y_{ij,t}}, \mathbf{y_{ij,p}})$ and losing all the valuable information regarding other samples contributes

Mean Absolute Error

This regression loss function is very similar to MSE.

Its expression is

$$\mathrm{MAE}(\mathbf{y_t}, \mathbf{y_p}) = \frac{\sum_{i=1}^{N} \sum_{j=1}^{d} |\mathbf{y_{ij,t}} - \mathbf{y_{ij,p}}|}{N \cdot d} \tag{3.52}$$

The only difference with MSE is that this loss "punishes" linearly any error. Therefore the slope of MAE is constant, while MSE is flattening around when close to its minimum.

The choice between MSE or MAE could be reformulated as: is a very small error tolerable or not? Even if using MAE does not imply a higher probability of reaching the minimum, it depends of course on the loss which best fits the problem.

There exist also other variants such as Mean Absolute Percentage Error, or Mean Squared Logarithmic Error which will not be treated, but their names are quite expressive of how they should look like and behave.

Huber

This regression loss function can be seen as a trade off between MSE and MAE.

Its expression is

$$\mathrm{HB}(\mathbf{y_t}, \mathbf{y_p}) = \frac{\sum_{i=1}^{N} \sum_{j=1}^{d} x}{N \cdot d} \quad \text{where } x = \begin{cases} \frac{|\mathbf{y_{ij,t}} - \mathbf{y_{ij,p}}|^2}{2} & \text{if } |\mathbf{y_{ij,t}} - \mathbf{y_{ij,p}}| \leq \delta \\ \frac{\delta^2}{2} + \delta(|\mathbf{y_{ij,t}} - \mathbf{y_{ij,p}}| - \delta) & \text{if } |\mathbf{y_{ij,t}} - \mathbf{y_{ij,p}}| > \delta \end{cases} \tag{3.53}$$

where δ is a free parameter which acts as a switch from MSE-like behaviour to MAE-like one. Calling ϵ_{ij} the error represented by $|\mathbf{y_{ij,t}} - \mathbf{y_{ij,p}}|$, it is possible to appreciate that the first derivative of HB with respect to ϵ_{ij} is equal to d for $\epsilon_{ij} > \delta$ and to ϵ_{ij} itself for $\epsilon_{ij} \leq \delta$, resulting in being continuous for $\epsilon_{ij} = \delta$.

Therefore HB can be interpreted in the following way: starting from $\epsilon_{ij} = 0$ we have a quadratic behaviour (typical of MSE), when we reach $\epsilon_{ij} = \pm\delta$ we lock the slope to

its current value $\pm\delta$ and we have a linear trend (typical of MAE). Doing so we have a continuous derivative and a tradeoff between MSE and MAE, resulting in a loss function which is much more robust to outliers (since they will fall in the MAE-like behaviour) and at the same time gives little importance to little errors (since they will fall in the MSE-like trend).

Cosine Similarity

This regression loss function is pretty different from all the previous ones. For this particular loss function we will interpret $\mathbf{y_{i,t}}$ and $\mathbf{y_{i,p}}$ as two vectors in an inner product vector space (usually $\mathbb{R}^n$ combined with the classic euclidean dot product $< \cdot\,;\cdot >$). The Cosine Similarity (CS) is simply represented by the cosine of the angle between these two vectors in such space.

Its expression is:

$$\mathrm{CS}(\mathbf{y_t}, \mathbf{y_p}) = \sum_{i=1}^{N} \frac{\cos(\theta_i)}{N} = \frac{1}{N} \sum_{i=1}^{N} \frac{< \mathbf{y_{i,t}}; \mathbf{y_{i,p}} >}{||\mathbf{y_{i,t}}|| \cdot ||\mathbf{y_{i,p}}||} = \frac{1}{N} \sum_{i=1}^{N} \frac{\sum_{j=1}^{d} \mathbf{y_{ij,t}} \cdot \mathbf{y_{ij,p}}}{\sqrt{\sum_{j=1}^{d} \mathbf{y_{ij,t}}^2} \cdot \sqrt{\sum_{j=1}^{d} \mathbf{y_{ij,p}}^2}} \tag{3.54}$$

By the way, we must notice how $\mathrm{CS}(\mathbf{y_t}, \mathbf{y_p}) \in [-1, 1]$, but it reaches its maximum value for $\mathbf{y_t} = \mathbf{y_p}$. While all of the other loss functions needed to be minimized, CS need to be maximized in order to give us the best outcomes. Therefore we have two options: changing the external algorithms in order to find the maxima of the loss function, or using -CS instead of CS. Since the second option is much more easier this will be the one implemented by most of the libraries.

Moreover we must pay a particular attention to null vector, which are orthogonal by definition to any other vector in the vector space. For such reason, CS will always output 0 if one of its 2 arguments is a null vector. This results in the uselessness of using CS as loss function if one of the true targets vectors can be a null vector, because it will end up being untrainable and hence useless, or even counterproductive.

3.2.11 Metric

Most, if not all, of the loss function could be used also as metric function. The difference between losses and metrics stands in their purpose: losses are used to train the CNN, while metrics are only used to evaluate CNN's performances.

Basically, we should choose a metric (or a set of metrics) based only on the trait of the performances that we wish to evaluate. Then we should decide metrics based on the problem we are tackling and the CNN's purpose, and since they do not affect the training there is no tuning for the metrics section.

Metrics are, by the way, the values we should use in our tuning process. A considerably better metric value (where better can be higher or lower, depending on the metric) could imply that the last set of hyperparameters could be the more adapt in achieving the required task. Anyway, in the comparison, many other factors should be considered. Were the compared CNNs trained for a reasonably number of iterations? Were they trained for around the same number of iterations? Are the results different enough, or

could the difference just be due to random fluctuations? Were the two CNNs evaluated on the same validation dataset?

Many of these questions can never be answered without doubts, here it lies one of the hardest point of training a whole CNN.

3.2.12 Batch Dimension

The Batch is the portion of training data which is considered before computing the updates through the optimizer.

It could seem that this number should have no effect on the the training itself, we always end up using all of the training data, just performing many small steps instead of one big step should not vary that much.

The importance of batch size is given by the presence of the derivative of the loss function in the updates computation. In fact, this derivative is evaluated based on the data we are using, so a small batch will have a lower probability to originate a gradient which is pointing in the direction of maximum decreasing slope.

To try to make that a little more clear, we could state that

$$\lim_{\eta \to 0^+} f(\mathbf{p_n}, X, y(\eta)) = f(\mathbf{p_n}, X, y_{true}) \tag{3.55}$$

where we use η to indicate noise and so $y(\eta)$ to indicate the targets we possess, which are affected by noise.

If we use a large number of samples, the effect of noise should average out, resulting in a much more reliable estimation of $\nabla(f(\mathbf{p}, X, y))$. By the way, in most of the cases we have an exact number of data samples, and it is impossible (or maybe just very hard due to cost, time, availability) to obtain more data.

Then, why should we choose to reduce on purpose the amount of data samples used to compute the loss gradient? The answer is that, doing so, we are more influenced by noise, and this stochastic effect gives a higher chance to avoid local minima or even escape from them.

On the other hand, reducing too much the batch dimension can result in a very noisy path of the updates, slowing down learning or even avoiding its convergence.

Therefore dataset dimension (and its quality) shall always be considered when choosing the batch dimension.

The most common options are the following:

- **Full-batch**, where the batch size is equal to the train dataset.
- **Mini-batch**, when the batch size is bigger than 1 data sample but smaller than the whole train dataset.
- **Online learning**, when the batch size is equal to 1 data sample.

In the last case, we are updating the vector $\mathbf{p}$ after every sample, thus resulting in the noisiest gradient possible.

Usually the option of the Mini-batch is chosen, since it represents the trade off and it is slightly tunable depending on the needs. The most common choices for the batch

size are powers of 2, since most of the runtime accelerators (GPUs and TPUs) split the data samples in 2^k subsets when running in parallel a training algorithm.

The remaining option of Full-batch is useful when the dataset is small, while the online learning option can be a valid option when we are running our training on a real-time training, with the possibility to perform training on a single data sample without even storing it.

Chapter 4

The Analysis

4.1 Introduction

Now it is time to focus on the main portion of this work. In order to investigate detectable signs of biological ageing we will consider a database of PPG signals, process such signals in order to clean them and perform feature extraction, identifying the most useful group of features through a procedure of feature selection. Finally we will compare the performances of our most useful group of features with the performances of the best CNN, in order to compare the Statistical Learning approach with the Deep Learning one.

Please notice that most of this chapter is extracted and elaborated from the BioRχve paper [27] of which professor Castellani and me are among the co-authors, and which contains the official outcomes of the reference study.

4.2 The Data

4.2.1 Data Acquisition

It is known that ML and mostly DL approaches often require big data availability. Given the nature of the study, we could not rely on small clinical trials from local hospitals. Therefore we obtained our data from the Heart for Heart initiative [28], a project which aims to become the world's largest heart health initiative. H4H data are PPG signal acquired through smartphones by using a specific app. The H4H initiative is powered by many partners, among which Happitech, the World's first smartphone CE certified heart rhythm SDK, for iOS and Android [29], with the aim of detecting heart rhythm disorders.

The PPG acquisition happens via the smartphone camera. By gently pressing your fingertip on top of the camera lens and partially covering the flash, the smartphone will behave like a reflection-mode photoplethysmograph, measuring the amount of light reflected by the fingertip and deducing the blood vessels' volume trend over time.

Of course this kind of data acquisition contains many more possible point of failure with respect to clinical trials. For a proper measurement, the smartphone app requires the subject to:

- put the hand on a firm surface,

- avoid moving either the hand or the phone (also moving in general, a different body posture can cause a different blood pressure in the fingertip and therefore a different blood volume not caused by the heart),

- avoid talking or taking deep breath (because talking, yawning and heavy breathing can affect PPG measurements, mostly in the low frequencies component [30].)

- wait around 100 seconds for calibration and measurement.

The drawback is that people are not monitored by any expert during the acquisition, so they could break one of the above suggestions and lower the measurement quality.

On the other hand, by having thousands of long records ($\sim 90s$ each) we will rely on the possibility to filter data quality and perform outliers detection.

After the measurement, some information is acquired from the subject, such as smoker status (a boolean parameter), height, sex, weight, etc, and most importantly the measurement, together with its information, is completely anonymized and only then it gets stored in the database.

4.2.2 Description of the used dataset

The database used for this study counts 4769 individuals. For each subject, one file in the .csv format is provided, containing the PPG recordings and some additional information. These additional information consisted of: sex, height, weight, and smoke (a boolean status for active smokers). The PPG recording is organized in 7 columns: time (around 90 seconds of measurement with a sample frequency of averagely 30 points per second), simultaneous recordings of red, green and blue reflected light (consisting in the PPG measurement), and the X, Y, and Z components of a three axis accelerometer.

Even if infrared/red light is more susceptible to motion artifacts with respect to the green component [31, 32, 33] we considered only the red component because of its higher amplitude with respect to the green and blue components, always relying on possible further outlier detection in cases of excessive measured motion artifacts.

4.3 Preprocessing

We first needed to remove the trend from the raw red PPG signal. By "removing the trend" we mean "removing the frequencies in the signal spectrum which are clearly lower than the interesting phenomena". In Figure 4.1 we can appreciate the visible distinction between higher frequencies (due to blood vessels pulsations) and lower frequencies (slower vertical shifts over time, which can be due to motion, consistent change in blood pressure, talking/heavy breathing). The second one is not directly linked to vascular phenomena, therefore we needed to remove it.

In order to do so we computed a centered moving average (CMA), which will contain only the frequencies below a certain threshold, and subtracted it from the raw signal, in order to perform a high-pass filter. The sliding window width $\mathbf{w}$ was made equal to the average sampling frequency of the signal, in this way we had around the same number of points per second for each of the different signals.

After that we needed to account for motion and noise artifacts. So we performed signal demodulation. We used the detrended signal (a real-valued signal s) and its Hilbert transform ($\mathcal{H}(s)$) to generate the analytic signal (a complex-valued signal $s + \mathcal{H}(s)$). The module of the analytic signal represents the instantaneous amplitude (also called the envelope) of the signal. This last one was obtained and then smoothed with another CMA with $w_2 = \frac{w}{2}$, which in this case was acting as a low pass filter. Lastly, the detrended signal was divided by the smoothed envelope, resulting in the detrended, demoduled, and denoised signal used for further analysis (Fig. 4.2).

Therefore, an accurate representation of the preprocessing python code for a single signal could be the following:

- We start with a raw signal raw consisting of the acquisition times sequence raw_{Time}, and 3 time series (respectively the red raw_R, green raw_G and blue raw_B reflected light components), with all of the 4 sequences of equal length.

- Since the emitted light came from a white flash, all the 3 light components should carry approximately the same information, with the only difference in Signal to noise ratio, due to the fact that some wavelengths are more absorbed by the tissues. Therefore we keep only the red component (raw_R), since it is the one with higher amplitude and better signal to noise ratio.

- We compute the average sampling frequency sf as

$$sf = \frac{\text{Nr. points in } raw_{Time}}{\max(raw_{Time}) - \min(raw_{Time})} \tag{4.1}$$

  where the usage of $\min(raw_{Time})$ is needed, because the acquisitions do not begin exactly at $Time = 0$.

- We compute the half-width w of the window that we will further use for the central moving average as

$$w = \lfloor \frac{sf}{2} \rfloor \tag{4.2}$$

  where $\lfloor \cdot \rfloor$ is the floor rounding operator, and the division by 2 is in reality a multiplication times 0.5 (because multiplications are computationally faster than divisions). Doing so, we will approximately consider $2w+1$ points in the computation of the central moving average, and recalling that $w = \lfloor sf/2 \rfloor$ we obtain that the number of points considered for the moving average is $\approx sf$ for $sf > 10$. In this way we have a total time width for our moving average of ≈ 1 second, and since for our data usually $sf \approx 30Hz$ the approximation can be considered as reliable.

- At this point we compute the CMA by using a "trick", in fact we achieve such result by convolving a box of $2w+1$ width and unitary area with the red component raw_R, giving us

$$\text{CMA}(raw_R, w) = raw_R * \frac{\overbrace{[1, 1, 1, \ldots, 1, 1]}^{2w+1 \text{ elements}}}{2w+1}. \tag{4.3}$$

Thanks to this, we achieve a faster and vectorized way to perform

$$\mathrm{CMA}(raw_R)_i = \frac{\sum_{i=-w}^{+w} raw_R}{2w+1}. \tag{4.4}$$

Please notice that the *total* width of the moving average is given by $2w+1$, which is always an odd number since w is an integer. This fact allows us to always have a unique central point for the window, which is what guarantees that we are performing a *central* moving average. We want to clarify that the boundary conditions are treated in a "valid" mode, which means that we will discard any point that will not have at least w points before and w points after it.

- As above explained, the CMA computation involves the removal of $2w$ points, the first w and the last w, from both raw_R and raw_{Time} as this points are ignored in the computation of the CMA

$$raw_{R,cut} = raw_R[w : -w] \tag{4.5}$$
$$raw_{Time,cut} = raw_{Time}[w : -w]. \tag{4.6}$$

- At this point, having the lower frequency components in the computed CMA, to obtain a high pass filter we simply subtract it from the original raw component. This gives us the detrended signal

$$detrended_R = raw_{R,cut} - CMA(raw_R, w) \tag{4.7}$$
$$detrended_{Time} = raw_{Time,cut}. \tag{4.8}$$

- now we can compute the analytic signal from the detrended signal using

$$analytic = detrended_R + i\mathcal{H}(detrended_R) \tag{4.9}$$

where $\mathcal{H}()$ indicates the Hilbert transform. Please notice that on the used python library (*scipy*) the function *scipy.signal.hilbert(x)* does not compute the Hilbert transform, but directly the analytic signal.

- Now we have the analytic signal, this is a complex-valued signal, so each of its elements will be characterized by an amplitude and a phase. Computing the absolute value of the analytical signal we recover the instantaneous amplitude, also called *envelope*, of the signal

$$envelope = |analytic| \tag{4.10}$$

- None of the previous operation has further shortened the arrays, therefore we compute the new average sampling frequency as

$$sf_{cut} = \frac{\text{Nr. points in } raw_{Time,cut}}{\max(raw_{Time,cut}) - \min(raw_{Time,cut})} \tag{4.11}$$

.

- Now we will smooth the envelope in order to remove very high frequencies (denoising) by computing its CMA with half-width of $w_{env} = \lfloor sf_{cut} \rfloor$ as a low pass filter, hence

$$\text{CMA}(envelope, w_{env}) = envelope * \frac{\overbrace{[1, 1, 1, \ldots, 1, 1]}^{2w_{env}+1 \text{ elements}}}{2w_{env} + 1} \tag{4.12}$$

  and again this will imply a shortening in the signal of $2w_{env}$ points, half at the begin and half at the end, for both $detrended_R$ and $raw_{Time,cut}$ (we are not considering *envelope* because we will no further use it, but if we needed to use it we would have to short it too)

$$detrended_{R,second\ cut} = detrended_R[w_{env} : -w_{env}] \tag{4.13}$$

$$raw_{Time,second\ cut} = raw_{Time,cut}[w_{env} : -w_{env}]. \tag{4.14}$$

- The last step consists in dividing the the detrended shortened signal by the smoothed envelope to perform demodulation

$$demoduled_R = \frac{detrended_{R,second\ cut}}{\text{CMA}(envelope, w_{env})}. \tag{4.15}$$

- The preprocessing algorithm concludes by returning two arrays: $demoduled_R$ (which has also been detrended and denoised) and $raw_{Time,second\ cut}$. These tow arrays are exactly $2w + 2w_{env} + 2$ points shorter than the input arrays, and consist the time series which will be analyzed and which will be used for further features extraction.

Another appreciable reason for which we kept only the red component is that, the preprocessing algorithm works properly for time series. For this reason, we would have to repeat it three times (1 for each light component) in order to maintain all the input arrays. This would result in triplicating the preprocessing runtime, factor that we evaluated as an avoidable problem considering a database counting thousands of individuals and a little amount of information carried by the green and blue but not the the red light component array.

4.4 Peak detection algorithm

To identify the correct peak locations, we slightly modified an existing algorithm [34]. The current algorithm consists in:

- compute the CMA of the processed signal and align it to its signal;
- locate the maximum of any region of the signal which is continuously above its moving average;
- label the set of identified maxima as partial peaks;
- repeat with the window width $\mathbf{w_i} = (0.5, 1, 1.5, 2, 2.5, 3) * sf$ (with sf = average sampling frequency), and consider as possible maxima those points labelled 6 times as partial peaks;

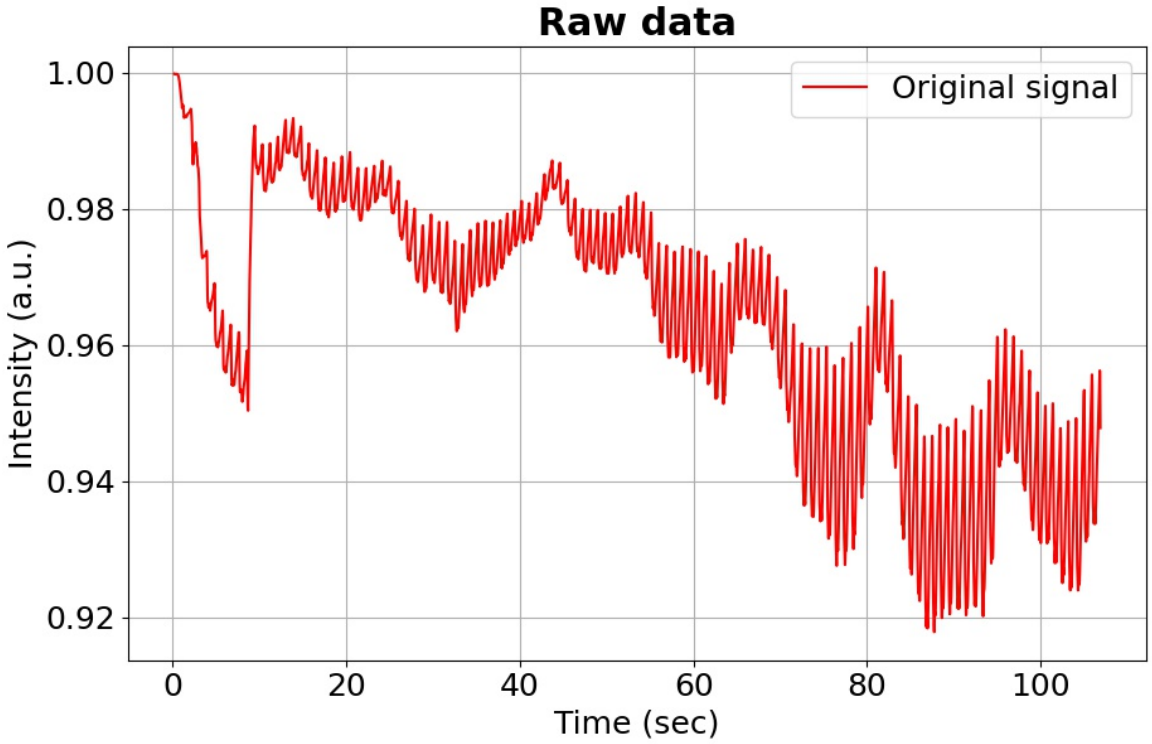

Figure 4.1: Example of used PPG signal, the high frequency variation of this kind of signals are linked to heart beats, while low frequency variations are probably related to talking/heavy breathing/moving of the patient.

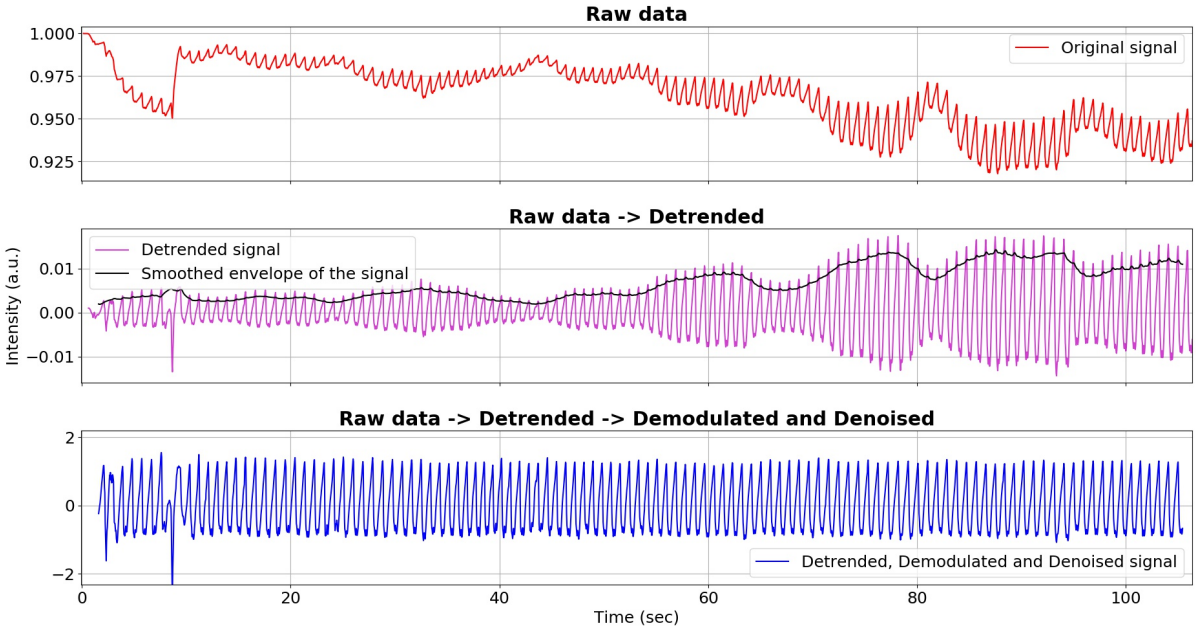

Figure 4.2: Preprocessing of PPG: (from top to bottom) raw, detrended, and demodulated and denoised clean signal.

- if two possible maxima are separated by less than 400ms, discard the smaller of the two.

If we assume that the delay from ventricular depolarization (indicated by the R-wave in the ECG) and the sphygmic wave reaching the fingertip (causing a peak in the PPG signal) is constant for different heart beats (its fluctuations are negligible with respect to its mean value), then we can compute from the PPG the time interval between heart beats. Having found the final peak positions, the array RR can be defined as the sequence of time intervals (expressed in ms) between consecutive detected peaks.

4.5 Feature extraction

In the following we will discuss the features extracted from the signal, grouping them by extraction field.

Features extracted from RR

Recalling that RR is a sequence of time intervals between consecutive peaks, from RR we can extract many information about heart rate variability (HRV) and inter beat time intervals distribution:

- ***ibi*** (average inter beat interval);
- ***medianRR***;
- ***madRR*** (median absolute distance of RR);
- ***sdnn*** (the standard deviation of RR, also called *standard deviation normal to normal*);
- ***tpr*** (turning point ratio of RR: computed as number of local extrema divided by number of points in the signal, ranging from 0 to 1 and used as and index of randomness for a given signal [35];
- ***skewnessRR***;
- ***kurtosisRR***; ***entropyRR*** (the Shannon entropy of the RR signal, considering $p_i = \frac{RR_i}{\sum RR_i}$).

Features extracted from $\mathbf{RR_{diff}}$

Computing the array RR_{diff} as

$$RR_{diff_i} = |RR_{i+1} - RR_i| \tag{4.16}$$

this one turns out being the array of absolute differences between consecutive elements of RR. The features derived from RR_{diff} are the following:

- ***sdsd*** (standard deviation of successive differences);
- ***rmssd*** (root mean square of successive differences);

- ***pnn20*** (proportion of normal to normal $>$ 20 ms: how many elements of RR differs from the previous more than 20ms, which can be reformulated as how many elements of RR_{diff} are larger than 20 ms);
- ***pnn50*** (same but for 50ms);
- $\boldsymbol{tpr}_{diff}$ (tpr computed on RR_{diff}).

SDPPG features

The second derivative of the PPG processed signal (SDPPG) has been linked to chronological age [36]. During each heart beat cycle the 5 typical points (***a*** - ***e***) can be identified (Fig. 4.3). To identify these points we computed the FDPPG (fourth derivative of PPG) for locating the zero crossings, which correspond to the inflection points in the SDPPG. Once two consecutive inflection points are found, only one local extreme can exist between them. The first and highest maximum of a beat cycle can be determined as ***a***, and the subsequent points, (***b***, ***c***, ***d*** and ***e***) can be detected as next minima or maxima starting from the previous one (As can be seen in Figure 4.3). After identifying these points, the features obtained from SDPPG are given by:

- the amplitudes;
- the time distances between any two of these points;
- the slope of the line segments connecting any two of these points.

Quality thresholding

A quality score (Q) of the signal was computed by taking into account bad demodulation (through the variance of detected peaks height) and noise (expressed as a number of local extrema that are not detected as peaks or their corresponding valleys):

$$Q = \text{Variance (detected height of peaks)} \times \frac{\text{Nr. local extrema} - \text{Nr. detected local extrema} + 2}{\text{Nr. points in the signal}}. \tag{4.17}$$

Please note that '+2' is used to avoid negative numbers in case of a detected peak having no corresponding valley before/after due to the length of the signal.

The Q score indicates how noisy and bad demodulated the signal is; the higher the score the poorer the quality, such that 0 is the perfect score. The threshold value of Q was set to 0.01. By applying quality thresholding after the feature extraction phase, a flexible choice of Q is possible without repeating the feature extraction step.

4.6 Data Analysis

4.6.1 Prediction of healthy vascular ageing (HVA)

After discarding incomplete data (e.g. missing age label) and quality thresholding with $Q < 0.01$, the cleaned database consisted of 3612 subjects: 2205 males and 1407 females, with an average age of 49 years and a standard deviation of 14.5 years.

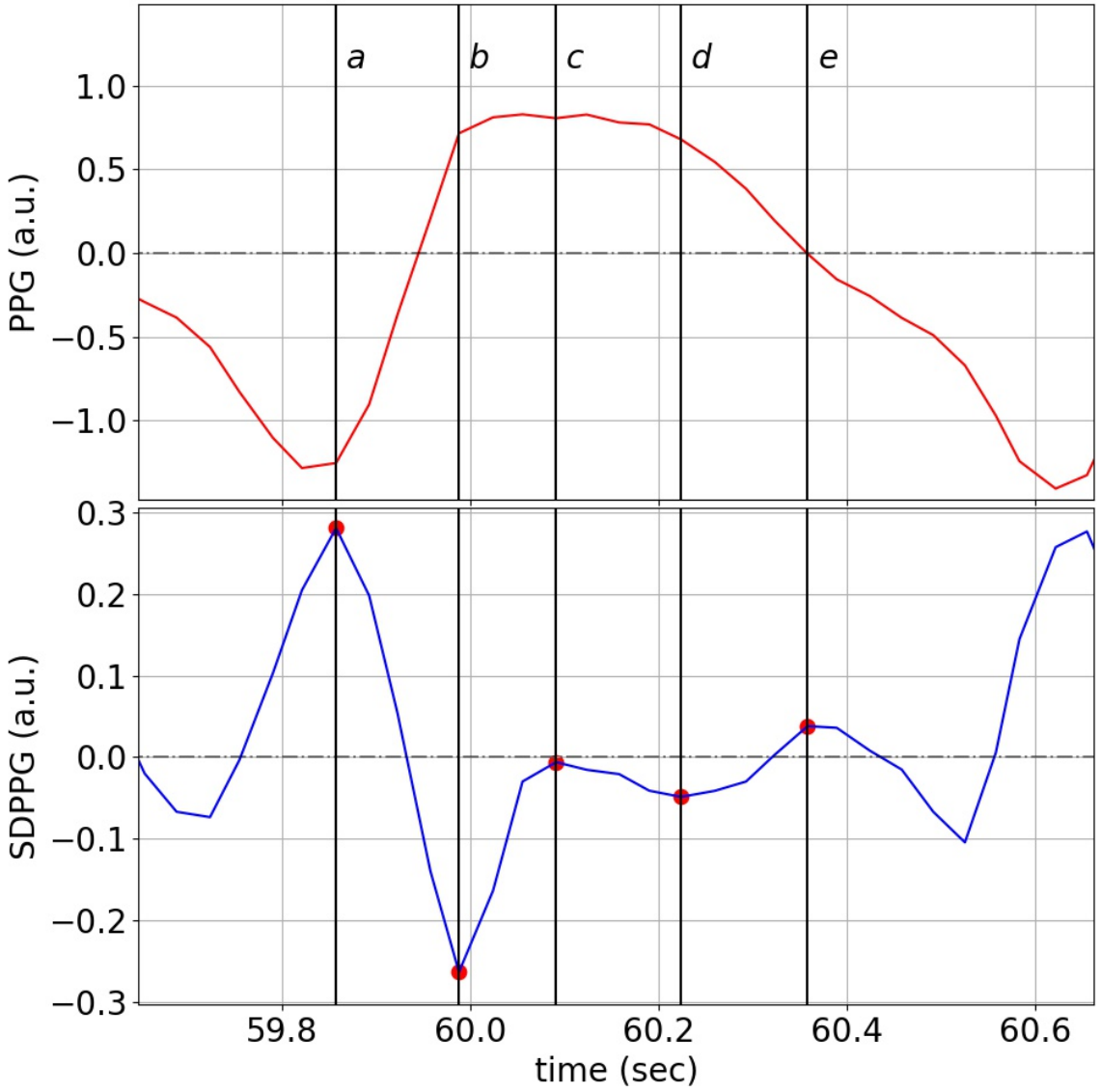

Figure 4.3: Comparison of PPG (top) and SDPPG (bottom) shapes. Red dots identify the ***a***, ***b***, ***c***, ***d*** and ***e*** points of SDPPG.

Next, the "continuous" outcome, age, was dichotomized into two classes: young (18 - 38 years, used as proxy for HVA) coded 1 and old (60 - 79 years, a proxy for non-HVA) coded 0.

By partitioning the data into a train (2709 subjects) and a test (903 subjects) set, we train and validate the model (both the ML and DL approaches) on a training set. This workflow is depicted in Figure 4.4.

The final model (ML) and hyperparameters (DL) were validated using the held-out test set. The age distribution of the training and test set was kept similar to that of the whole database thanks to a stratified splitting with respect to the target variable.

Ridge regression

Prior to the analysis, the 38 extracted features were robustly standardized by subtracting the median and dividing by the interquartile range. Since some features were highly

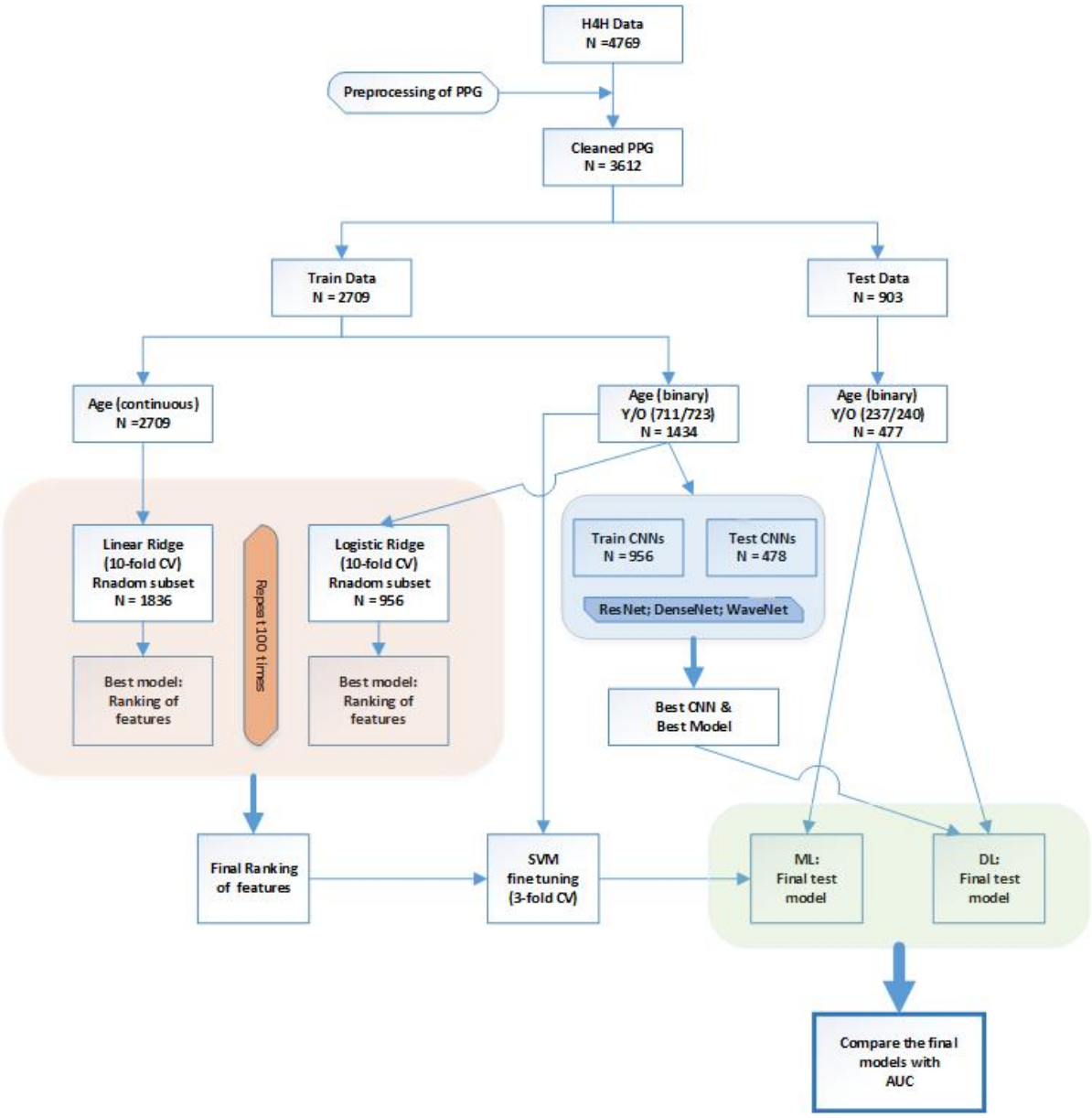

Figure 4.4: Workflow of PPG analysis to predict healthy vascular ageing (HVA) using ML and DL approaches.

correlated (dark red or blue colors in Fig. 4.5) and there may be multicollinearity issues, a penalized regression can be considered as much more useful. Shrinking the coefficient values towards zero in multiple regressions allows the less contributing variables to have a coefficient close or equal to zero.

To jointly select the relevant features for HVA, we applied ridge penalized regression; linear and logistic ridge regressions were employed for continuous and binary age outcomes, respectively [37, 38]. To obtain the final model, 2/3 of the training dataset were randomly sampled and used to fit the model; for fine-tuning of the penalty parameters a 10-fold cross-validation (CV) was applied. This procedure was repeated 100 times for both continuous age (using linear ridge) and for dichotomized age (using logistic ridge). The resulting rankings of the coefficients were recorded, and averaged over 100 runs. The

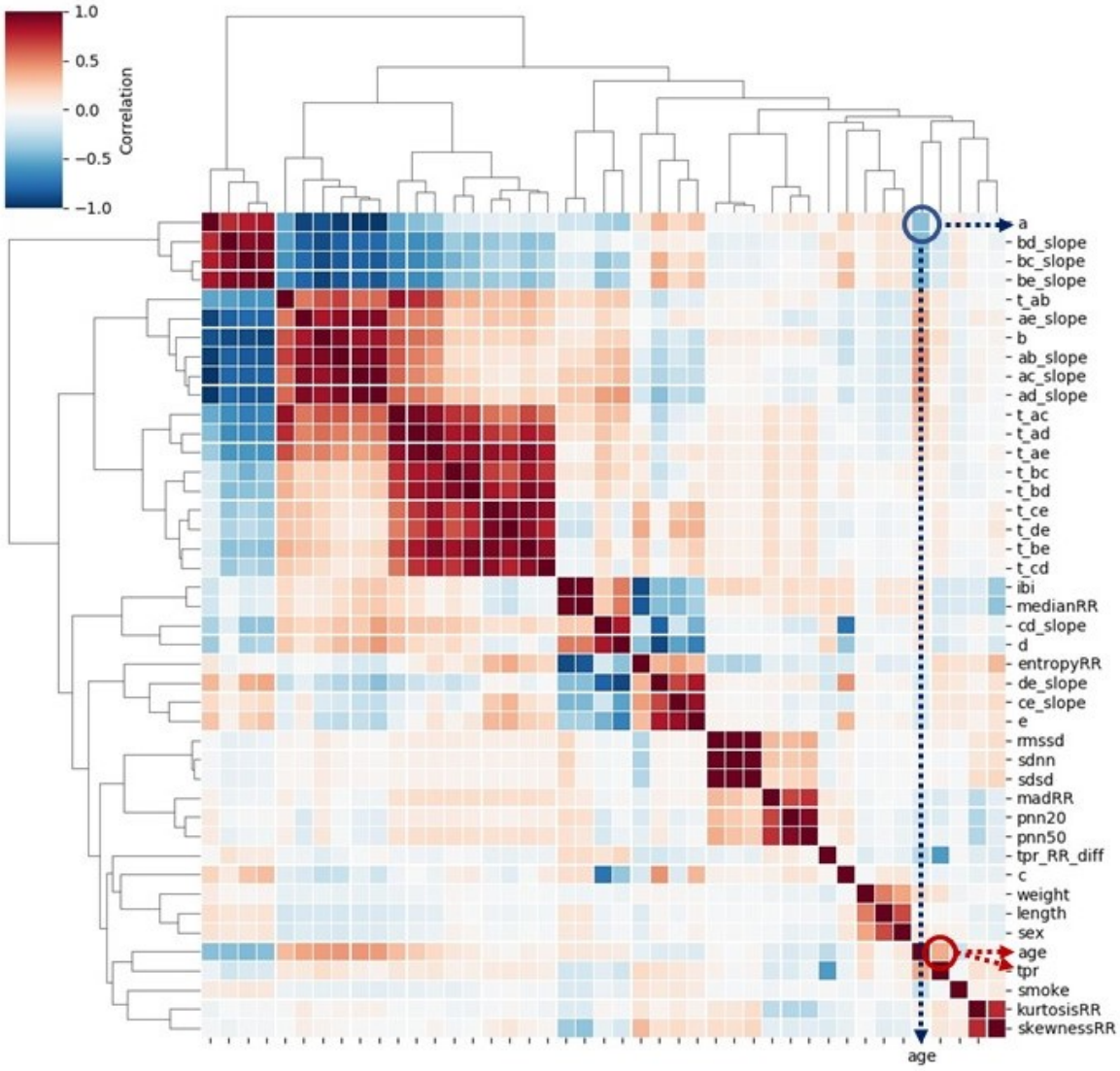

Figure 4.5: Heatmap of Pearson correlation between 38 features extracted from PPG, four covariates, and age: the feature ***a*** and age shows negative correlation, while ***tpr*** and age are positively correlated.

final score for the ranking was achieved by summing up the two scores. Then we selected some combinations of the most relevant features and we evaluated their HVA/non-HVA classification performance using a Support Vector Machine (SVM) classifier (previously tuned with a 3-fold CV on training set) on the test set.

Convolutional Neural Networks)

For the Deep Learning approach, we built and trained many convolutional neural networks (CNNs). For CNNs, the processed signals (not the extracted features) were used as input, and the dichotomized age as target. The reason behind the input choice was the will of avoiding the CNN to discover different features. By giving the whole signal as possible input to our CNNs we are allowing these ones to extrapolate information that

Hyperparameter	Compared options
kind of CNN	WaveNet, DenseNet, **ResNet**
Hidden layers number	4 8 **12**
dilation rate (only for WaveNets)	1 2 4 8
Batch dimension	4 16 **32** 64 256
Kernel dimension	3 6 10 30 **50** 80
Activation function	ReLU **ELU** SELU
Optimizer	SGD ADAM **NADAM**
Start learning rate	10^{-1} 10^{-2} 10^{-3} $\mathbf{10^{-4}}$ 10^{-5}
Patience	5 **8** 10

Table 4.1: Results from DL approach: different values of hyperparameters for CNNs (WaveNet, DenseNet, and ResNet). The hyperparameter patience indicates the number of epochs to wait before early stop if no improvement in the loss function is achieved. There is no optimal dilation rate value since it is only used for WaveNets, while the best model was a ResNet CNN.

we did not acquire or selected previously, such choice will be fundamental in the further comparison between Statistical Learning and Deep Learning approaches.

Among many CNNs possible structures we chose to considerate ResNets, DenseNets and WaveNets. The first two having the benefits of avoiding many gradient vanishing problems, by feeding directly any convolutional layer using the sum (ResNet) or the concatenation (DenseNet) of all the previous layers' output, respectively. The third because it was proposed to accommodate the specific nature of time sequences [39], by using a different kind of time-oriented convolution. Moreover, ResNet was found to be highly effective with some time series datasets [40], while DenseNet used on PPG signals has already shown good results [41].

the CNNs were trained, epoch by epoch, using 2/3 of the training dataset. Their performance was validated using the remaining 1/3 of the training set; for each hyperparameter, many different values were tried out and the best was determined (see Table 4.1, where the bold-faced ones show the hyperparameters of the best model based on the validation set performances).

Evaluation of the performance of ML and DL approaches

The optimized models by ML and DL were validated using the held-out test set. The prediction performances of five different models were compared using area under the curve (AUC): four covariates (weight, height, sex, smoking); the best two PPG features (***a*** and ***tpr***) based on ML; covariates and these two features; the best performing CNN; covariates and 38 PPG features.

All the algorithms and analysis have been performed using Python 3.7 and its suitable Anaconda distribution [42]. The code for preprocessing, feature extraction, ML and DL analysis can be found on GitHub, at `https://github.com/Nico-Curti/cardio`, and at `https://github.com/LorenzoDallOlio/vascular-ageing`.

4.7 Results

4.7.1 Exploratory data analysis

Firstly we exploited possible separation among chronological young and old. To do so, we used the spectral embedding [42] function for non-linear dimensionality reduction. This function uses the Laplacian Eigenmaps algorithm previously discussed in section 2.3. Figure 4.6 shows a relatively good separation between chronologically young (blue points, indicating HVA) and old (red points, indicating non-HVA). The same figure also shows a gradual transition from HVA to non-HVA.

To visualize how strong the correlations are among age, the four covariates (sex, weight, height, and smoking) and the 38 PPG features (extracted from the PPG signals), a heatmap based on Pearson correlation coefficient was computed (Figure 4.5). The strongest correlation between age and PPG features was found with:

- many elements of the SDPPG group, with maximum correlation for features ***slope-AC*** and ***a*** ($r \geq 0.40$ for both);
- the ***tpr*** ($r \geq 0.35$).

Since the group of SDPPG related features has a high internal multicollinearity, we would expect to have a couple of useful parameters in the extracted features, by combining one of the SDPPG features with the ***tpr*** feature.

We then investigated how well the classifier for healthy vascular ageing (HVA) would perform using all extracted PPG features as a reference point for further comparisons.

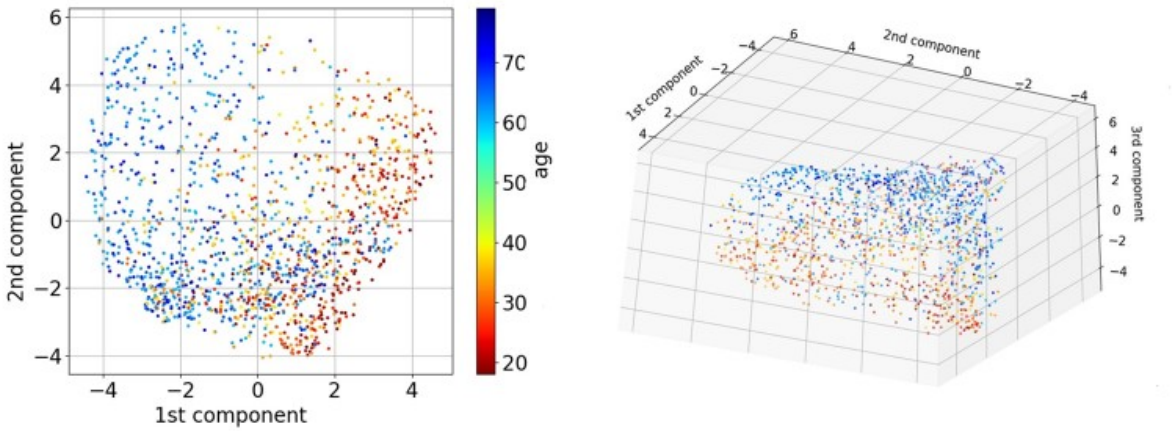

Figure 4.6: Dimension reduction of 38 PPG features extracted by spectral embedding: the left depicts the first two components, while the right depicts the first three components.

4.7.2 Application of ML and DL to predict HVA

The final scores achieved by linear and logistic ridge regression are reported in Table 4.2. The best performing PPG features (excluding the covariates) were ***a*** (from SDPPG) and ***tpr*** (from PPG).

Variable name	Linear ridge	Logistic ridge	Final ranking
smoking	1.89	1.76	3.65
sex	4.81	4.67	9.48
a	6.64	10.06	16.70
tpr	10.61	7.56	18.17
ibi	12.36	8.81	21.17
pnn20	14.72	7.65	22.37

Table 4.2: Results from ML approach: top six ranked variables from four covariates (smoking, sex, weight, and height) and 38 features extracted from PPG. The scores of linear and logistic ridge were achieved by ranking the absolute value of regression coefficients in decreasing order and then those rankings across 100 repetitions. The final score shows the sum of linear and logistic average rankings.

The expectation of the first exploratory analysis were attended, and therefore we selected the feature ***a*** from the SDPPG group as main candidate for being informative regarding the biological age.

4.7.3 Evaluation of prediction performance

In order to validate the results, the independent test set was used (Fig.4.4). The AUC obtained using only one feature (i.e., ***a***, ***slope-AC***, or ***tpr***) was around 0.8 (Table 4.3). The following models were considered: (i) covariates (weight, height, sex, smoking), (ii) the best two PPG features (***a*** and ***tpr***) from ML, (iii) covariates and these two features, (iv) the best performing CNN, and (v) covariates and all PPG features. One should take note that the model (v) is not recommended due to the many used features. So many features can slow down estimation procedure and more importantly cause overfitting, which may not lead to such a good performance on a different dataset. In order to compare the prediction performance of the five models, the area under the ROC curve (AUC) was computed, and the results are depicted in Figure 4.7. By adding the PPG features (***tpr*** and ***a***) to the covariates, AUC increased from 0.742 to 0.947. The 12-layer ResNet model (AUC=0.953) performed similar to this last model and similar to the model which was including all the variables (AUC=0.954), showing that most of the used information could be embedded by just these 2 features and 4 covariates.

4.7.4 Sex-stratified analysis

There are differences in body weight, height, body fat distribution, heart rate, stroke volume, and arterial compliance between the two sexes. In the very elderly, age related large artery stiffness is reported to be more pronounced in women [43]. Therefore, based on the two features obtained by the best performing ML, we investigated whether men and women age differently regarding HVA. The model, age = ***a*** + ***tpr*** + covariates + error, was used to estimate coefficients using the training set. The obtained coefficient estimates were used to predict vascular ageing using the test set. The results are shown in Figure 4.8. The points in the figures are the predicted age of each male (blue) and female (red). To depict the fitted trend line the locally weighted scatterplot smoothing

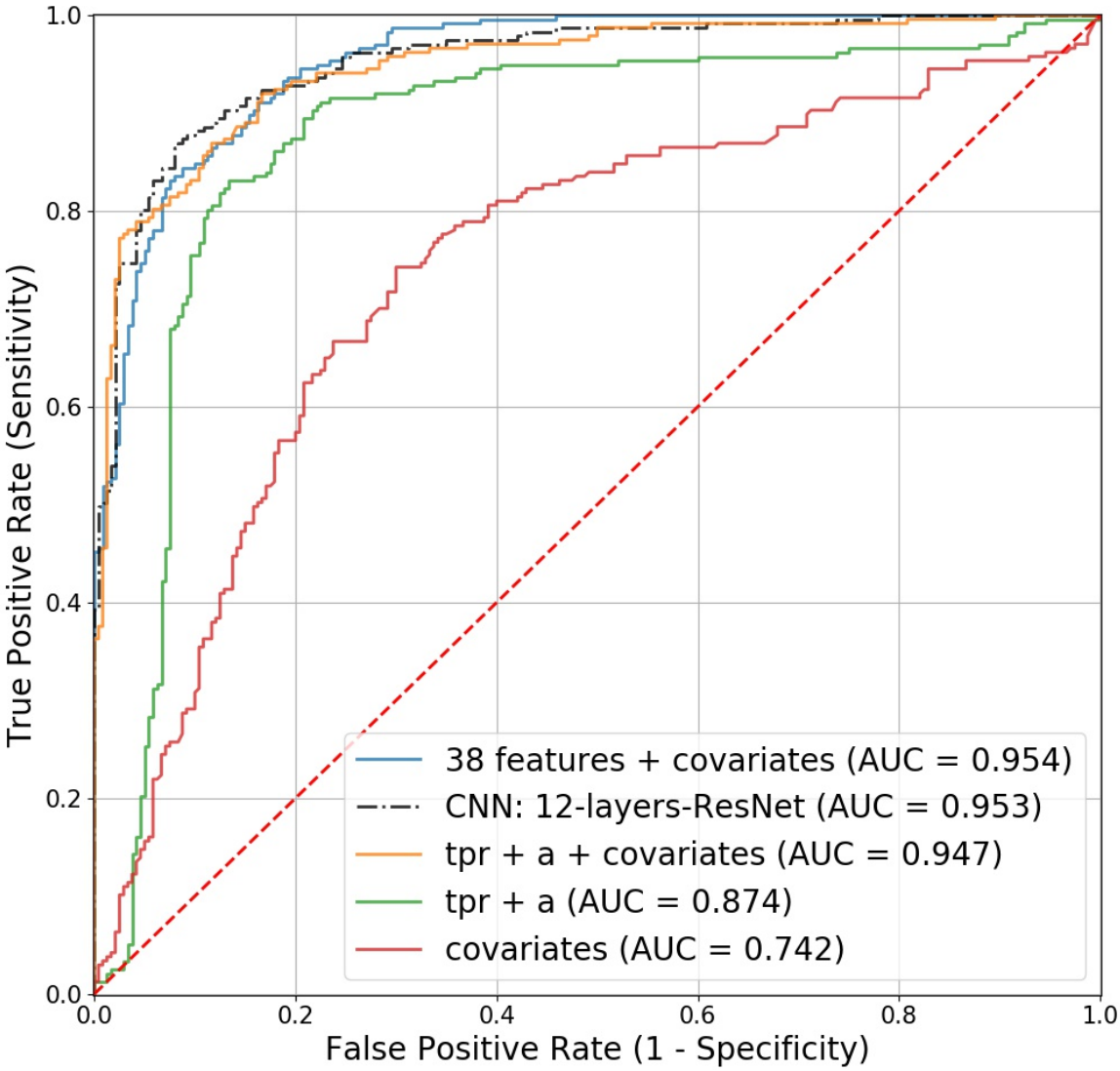

Figure 4.7: Receiver operating characteristic (ROC) curves for the competing models in predicting healthy vascular ageing (HVA): the four covariates include sex, weight, height, and smoking.

(LOWESS) technique was used. For clear comparison, two fitted lines are presented (in right panel). In average, females are healthier than males regarding vascular ageing. Focusing on the individual feature, the $\boldsymbol{a}$ wave from SDPPG showed the considerable difference between men and women (Figure 4.9).

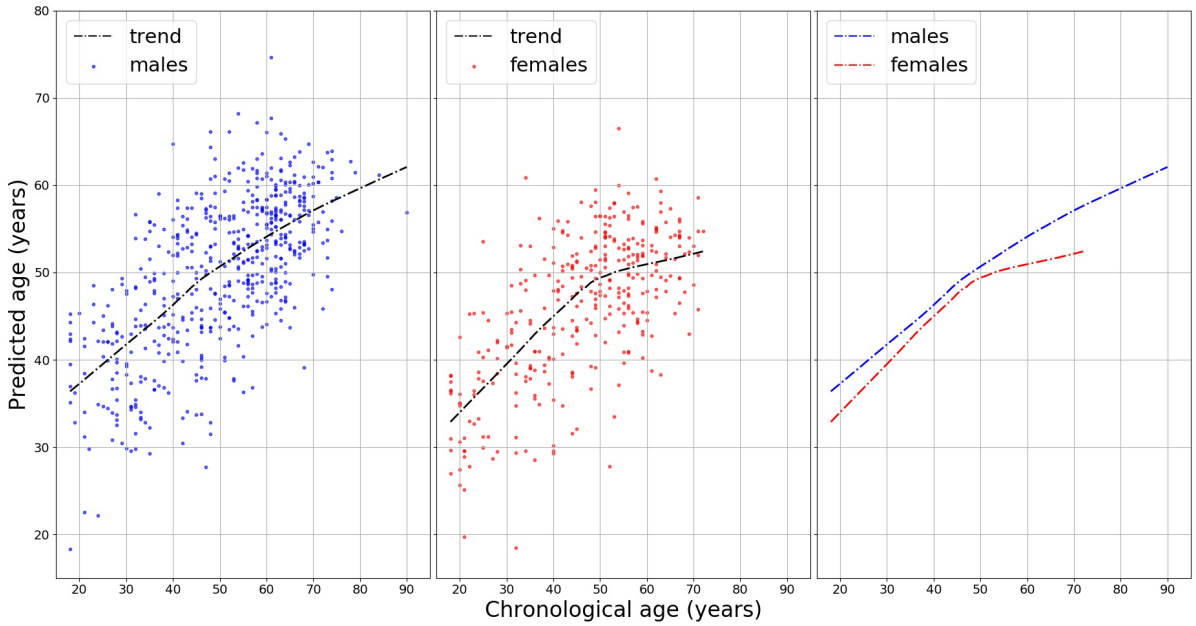

Figure 4.8: Sex-stratified analysis for predicting vascular age: a locally weighted scatterplot smoothing (LOWESS) of the ages predicted by the model ***a***+***tpr***+covariates for men and women, separately. The figure in right panel shows that average predicted age for females is always lower than that for males, indicating females have healthier vascular ageing. Moreover, for females, a reduction in the trend slope around 50 years old is shown (in middle panel).

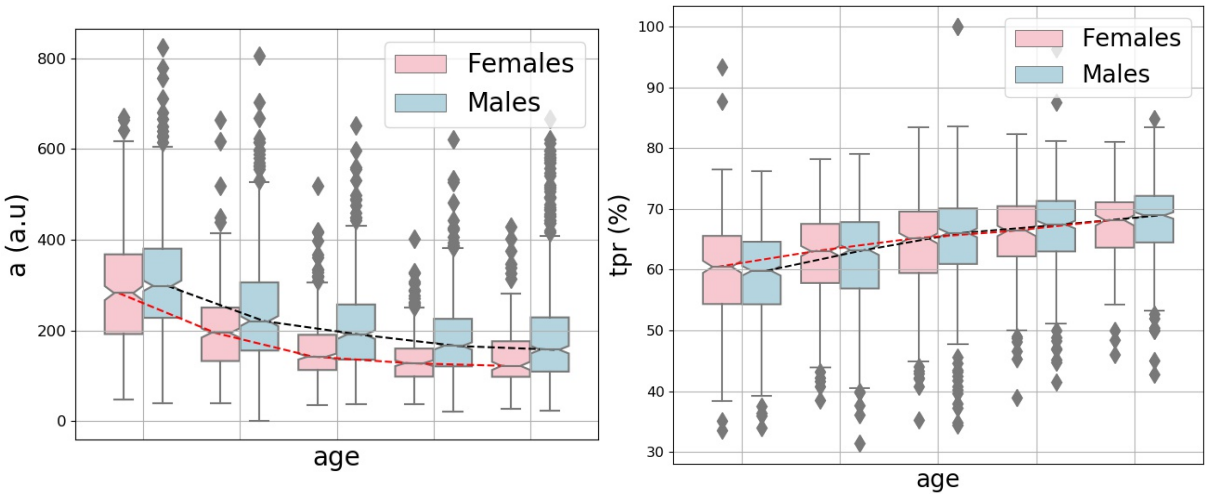

Figure 4.9: Box plots of age quintiles stratified by sex. Please note that ***tpr*** (right) increases with age, while ***a*** (left) decreases and is appreciably lower for females.

Model	Sensitivity (%)	Specificity (%)	AUC (%)
38 features + covariates	87.76	85.42	95.43
CNN: 12-layers-ResNet	90.72	85.83	95.34
a + tpr + covariate	87.34	86.25	94.73
ac_slope + tpr + covariates	87.76	88.33	94.60
a + tpr	83.97	82.50	87.43
ac_slope	78.06	78.75	81.68
a	77.64	78.33	80.79
tpr	73.00	76.25	80.50
covariates	69.62	72.50	74.21
pnn20	66.67	61.67	66.29
ibi	40.93	72.50	55.21

Table 4.3: Sensitivity, Specificity and AUC scores for some of the compared models.

Chapter 5

Conclusions

We conclude this work with an overall outcome analysis. The aim of the study was to exploit the biological ageing phenomena which affects the blood vessels, and at the same time to compare the outcomes of a Statistical Learning approach with a Deep Learning approach. The first part could be considered as achieved, as soon as the evaluation of generalization performance led to a good estimation of vascular ageing. The second aspect can be appreciated in the comparison of deep convolutional neural networks with a features extraction procedure. Even if the two approaches led to similar performances, the explainability and higher repeatability of the Statistical Learning make it preferable to the Deep Learning one, which acts much more as a "black box method" by its nature.

Moreover, our work was motivated by the unique crowd-sourced data from Heart for Heart initiative. Publicly available PPG data that includes 'raw' signals are still highly rare. Having abundant PPG data from the general population, we aimed to scrutinize the whole process from data preprocessing of the PPG signals up to the prediction of HVA using PPG.

Recently deep neural network approaches have grown in popularity [41, 44], there are several challenges applying DCNN (deep convolutional neural network) to health data. One needs to choose the right topology of NN: how many hidden layers, how do you trade off the number of parameters versus the amount of training data, etc. Computational power is one of the biggest limitation, when the aim is to get instant diagnosis using a wearable device [45]. Moreover, the algorithms deployed are inscrutable (as we already said, DL is a "black-box" approach) and it is difficult to interpret their results compared to other ML approaches. We employed several steps before the application of ML: the signals was detrended and demodulated, the quality of the signals was assessed and signals of poor quality removed, peak detection algorithm was performed, and then relevant features extracted.

Instead of using some aspects extracted from the signal, the whole signal was used as input for DL; the latter might be advantageous, when certain hidden features were not extracted and included in the model for ML. Therefore, we have investigated effectiveness of applying the computer intensive DL methods (ResNet, DenseNet, WaveNet) against the relatively simple ML method (ridge penalized regression).

The prediction performance indicated the best DL (AUC of 95.3%), 12-layers ResNet, performed slightly better than the ML (AUC, sensitivity, and specificity of 94.7%, 87.3%, and 86.3%) with the model, 2 PPG features ($\boldsymbol{a}$ and $\boldsymbol{tpr}$) + 4 covariates (sex, weight, height, smoking).

Nevertheless, ML had the merit of identifying potential biomarkers. It has been reported that features derived from the contour of PPG signals showed association with age, in particular regarding arterial elasticity and the changes in the elastic properties of the vascular system [46]. The feature ***a***, which is the amplitude of a wave extracted from SDPPG and ***tpr*** from PPG showed negative and positive correlation with age, respectively.

The SDPPG features derived from the amplitudes of the distinctive waves situated in the systolic phase of the heart cycle, quantify the acceleration of the arterial blood vessels' walls. From a lower acceleration we can deduce lower deformation of such blood vessels' walls, and then we can link this to a lower elasticity (or a higher stiffness) due to the ageing process. ***tpr*** (the turning point ratio or the RR array) is based on the non-parametric 'Runs Test' to measure the randomness in a time-series, and it is higher when series are more random: for instance, patients affected by atrial fibrillation tend to have higher tpr than healthy patients ***tpr*** [35]. This fact is somehow confirming what we have found. In fact, we would expect to have both unhealthy and older patients, different from healthy patients, suggesting again that the biological ageing process is some sort of measurable degradation, with effects comparable to a disease.

Taking age as a proxy for vascular ageing, both larger ***a*** and lower ***tpr*** predict healthy vascular ageing (HVA). In addition vascular ageing differs between the sexes. Age-related changes in vascular function generally include increasing endothelial dysfunction and arterial stiffness [47]. We have shown that women appeared to be healthier (younger) than men regarding to vascular ageing (Figure 4.8). Moreover, the clear slope variation in the females related plot of Figure 4.8 might be interpreted as a slowdown in the vascular ageing process, ultimately leading to longer life expectancy of women. The best performing ML model (***a*** + ***tpr*** + sex + weight + height + smoking) supports these findings.

Therefore, for predicting HVA, ML using a few PPG features together with relevant clinical information can be seen as a viable option against computer-intensive DL approaches [45]. Further, this study might be considered as a proof of concept for prediction of vascular ageing based on chronological age. Extending and fine-tuning of the best performing ML model may lead to a risk score for vascular ageing. Adding other relevant variables (the known risk factors) into the current model can be an option to consider.

Taking PPG measurements by smartphone camera together with a simple algorithm for feature extraction and prediction of vascular ageing facilitates self-monitoring of individual risk score, which can be linked to smoking and eating habits.

Note that CNNs investigated here demonstrated good performances obtained by simple structures (never more than 12 hidden layers, which is feasible for common laptop to train). When accurate prediction is the main purpose and any distinct features need to be fully incorporated (as for atrial fibrillation detection), CNNs can be good candidates. Our results with age as a target could be furtherly employed for transfer learning approaches, such as for a specific target detection like the presence/absence of atrial fibrillation in PPG signals.

Last but not least, we showed that PPG measured by smartphone has the potential for large scale, non-invasive, patient-led screening. However, current evidence is often biased due to low quality studies, black-box methods, or small sample sizes. For instance, the reliability of ultra-short HRV features (PPG measurements less than 5 minutes) remains

unclear and many HRV analyses have been conducted without questioning their validity [48]. On the other end of spectrum, the systematic review of assessing the balance of benefits and harms of screening for atrial fibrillation with smartphone acquired PPG is lacking. This work contributes to establishing generally accepted algorithm based on open data and software, which is of major importance to reproduce the procedures, and to further improve and develop methods.

Bibliography

[1] Mohamad Alnaeb, Nasser Alobaid, Alexander Seifalian, Dimitri Mikhailidis, and George Hamilton. Optical techniques in the assessment of peripheral arterial disease. *Current vascular pharmacology*, 5:53–9, 02 2007.

[2] Mohamed Elgendi. On the Analysis of Fingertip Photoplethysmogram Signals. *Current Cardiology Reviews*, 8(1):14–25, June 2012.

[3] B Jani. Ageing and vascular ageing. *Postgraduate Medical Journal*, 82(968):357–362, June 2006.

[4] Gary E McVEIGH, Patrick B Allen, David R Morgan, Colm G Hanratty, and Bernard Silke. Nitric oxide modulation of blood vessel tone identified by arterial waveform analysis. page 7, 2001.

[5] Gary E. McVeigh, Dennis J. Morgan, Stanley M. Finkelstein, Lisa A. Lemay, and Jay N. Cohn. Vascular abnormalities associated with long-term cigarette smoking identified by arterial waveform analysis. *The American Journal of Medicine*, 102(3):227 – 231, 1997.

[6] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. Adaptive computation and machine learning series. MIT Press, Cambridge, MA, 2012.

[7] Andrew Y Ng, Michael I Jordan, and Yair Weiss. On Spectral Clustering: Analysis and an algorithm. page 8, 2001.

[8] Mikhail Belkin and Partha Niyogi. Laplacian Eigenmaps for Dimensionality Reduction and Data Representation. *Neural Computation*, 15(6):1373–1396, June 2003.

[9] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016. `http://www.deeplearningbook.org`.

[10] Ruslan Salakhutdinov and Geoffrey Hinton. Deep boltzmann machines. *Proceedings of AISTATS 2009*, 5:448–455, 01 2009.

[11] Haoning Lin, Zhenwei Shi, and Zhengxia Zou. Maritime Semantic Labeling of Optical Remote Sensing Images with Multi-Scale Fully Convolutional Network. *Remote Sensing*, 9(5):480, May 2017.

[12] Andrinandrasana David Rasamoelina, Fouzia Adjailia, and Peter Sincak. A Review of Activation Function for Artificial Neural Network. In *2020 IEEE 18th World Symposium on Applied Machine Intelligence and Informatics (SAMI)*, pages 281–286, Herlany, Slovakia, January 2020. IEEE.

[13] Yann LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Efficient backprop. In Grégoire Montavon, Genevieve B. Orr, and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade (2nd ed.)*, volume 7700 of *Lecture Notes in Computer Science*, pages 9–48. Springer, 2012.

[14] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. page 8, 2010.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *arXiv:1502.01852 [cs]*, February 2015. arXiv: 1502.01852.

[16] Ludovic Trottier, Philippe Giguere, and Brahim Chaib-draa. Parametric Exponential Linear Unit for Deep Convolutional Neural Networks. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 207–214, Cancun, December 2017. IEEE.

[17] Abien Fred M Agarap. Deep Learning using Rectified Linear Units (ReLU). page 7, 2019.

[18] Djork-Arne Clevert, Thomas Unterthiner, and Sepp Hochreiter. FAST AND ACCURATE DEEP NETWORK LEARNING BY EXPONENTIAL LINEAR UNITS (ELUS). page 14, 2016.

[19] Bin Ding, Huimin Qian, and Jun Zhou. Activation functions and their characteristics in deep neural networks. In *2018 Chinese Control And Decision Conference (CCDC)*, pages 1836–1841, Shenyang, June 2018. IEEE.

[20] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-Normalizing Neural Networks. *arXiv:1706.02515 [cs, stat]*, September 2017. arXiv: 1706.02515.

[21] Zhen Huang, Tim Ng, Leo Liu, Henry Mason, Xiaodan Zhuang, and Daben Liu. SNDCNN: Self-normalizing deep CNNs with scaled exponential linear units for speech recognition. *arXiv:1910.01992 [cs, eess, stat]*, March 2020. arXiv: 1910.01992.

[22] Hugo Larochelle, Yoshua Bengio, Jerome Louradour, and Pascal Lamblin. Exploring Strategies for Training Deep Neural Networks. *Journal of Machine Learning Research*, page 40, 2009.

[23] Ilya Sutskever, James Martens, and George Dahl. On the importance of initialization and momentum in deep learning. page 9, 2013.

[24] D. P. Kingma and L. J. Ba. Adam: A Method for Stochastic Optimization. 2015. Publisher: Ithaca, NYarXiv.org.

[25] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. ON THE CONVERGENCE OF ADAM AND BEYOND. page 23, 2018.

[26] Timothy Dozat. INCORPORATING NESTEROV MOMENTUM INTO ADAM. page 4, 2016.

[27] Lorenzo Dall'Olio, Nico Curti, Daniel Remondini, Yosef Safi Harb, Folkert W. Asselbergs, Gastone Castellani, and Hae-Won Uh. Prediction of vascular ageing based on smartphone acquired ppg signals. *bioRxiv*, 2020.

[28] Sudler & Hennessey. Heart For Heart. *Website* `http://www.heartrateapp.com/`, 2020.

[29] Happitech. Monitor your heart rhythm using only a smartphone. *Smartphone App* `http://www.happitech.com`, 2020.

[30] Hongwei Yuan, Sven Poeggel, Thomas Newe, Elfed Lewis, Charusluk Viphavakit, and Gabriel Leen. An Experimental Study of the Effects of External Physiological Parameters on the Photoplethysmography Signals in the Context of Local Blood Pressure (Hydrostatic Pressure Changes). *Sensors*, 17(3):556, March 2017.

[31] Toshiyo Tamura, Yuka Maeda, Masaki Sekine, and Masaki Yoshida. Wearable photoplethysmographic sensors—past and present. *Electronics*, 3(2):282–302, 2014.

[32] Kenta Matsumura, Peter Rolfe, and Takehiro Yamakoshi. iPhysioMeter: A smartphone photoplethysmograph for measuring various physiological indices. *Methods in Molecular Biology*, 1256:305–326, 2015.

[33] D Castaneda, A Esparza, M Ghamari, C Soltanpur, and H Nazeran. A review on wearable photoplethysmography sensors and their potential future applications in health care. *International Journal of Biosensors & Bioelectronics*, 4(4):195–202, 2018.

[34] Paul van Gent, Haneen Farah, Nicole van Nes, and Bart van Arem. HeartPy: HeartPy: A novel heart rate algorithm for the analysis of noisy signals. *Transportation Research Part F: Traffic Psychology and Behaviour*, 66:368–378, 2019.

[35] Sung-Chun Tang, Pei-Wen Huang, Chi-Sheng Hung, Shih-Ming Shan, Yen-Hung Lin, Jiann-Shing Shieh, Dar-Ming Lai, An-Yeu Wu, and Jiann-Shing Jeng. Identification of Atrial Fibrillation by Quantitative Analyses of Fingertip Photoplethysmogram. *Scientific Reports*, 7(3):45644, 2017.

[36] Kristjan Pilt, Rain Ferenets, Kalju Meigas, Lars-Göran Lindberg, Kristina Temitski, and Margus Viigimaa. New photoplethysmographic signal analysis algorithm for arterial stiffness estimation. *The Scientific World Journal*, page 169035, 2013.

[37] Arthur E Hoerl and Robert W Kennard. Ridge regression: biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

[38] Trevor. Hastie, Robert. Tibshirani, and J. H. (Jerome H.) Friedman. The elements of statistical learning : data mining, inference, and prediction. *Springer*, 2001.

[39] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alexander Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. WaveNet: A Generative Model for Raw Audio. *Arxiv* `http://dx.doi.org/10.6084/m9.figshare.853801`, 2016.

[40] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. Deep learning for time series classification: a review, 2019.

[41] Ming-Zher Poh, Yukkee Cheung Poh, Pak-Hei Chan, Chun-Ka Wong, Louise Pun, Wangie Wan-Chiu Leung, Yu-Fai Wong, Michelle Man-Ying Wong, Daniel Wai-Sing Chu, and Chung-Wah Siu. Diagnostic assessment of a deep learning system for detecting atrial fibrillation in pulse waveforms. *Heart*, 104(23):1921–1928, 2018.

[42] Anaconda. Anaconda Software Distribution. *Software* `http://www.anaconda.com`, 2016.

[43] B Jani and C Rajkumar. Ageing and vascular ageing. *Postgraduate Medical Journal*, 82(968):357–362, 2006.

[44] Soonil Kwon, Joonki Hong, Eue Keun Choi, Euijae Lee, David Earl Hostallero, Wan Ju Kang, Byunghwan Lee, Eui Rim Jeong, Bon Kwon Koo, Seil Oh, and Yung Yi. Deep learning approaches to detect atrial fibrillation using photoplethysmographic signals: Algorithms development study. *JMIR Mhealth Uhealth*, 21(6):e12770, 2019.

[45] Syed Khairul Bashar, Dong Han, Shirin Hajeb-Mohammadalipour, Eric Ding, Cody Whitcomb, David D. McManus, and Ki H. Chon. Atrial Fibrillation Detection from Wrist Photoplethysmography Signals Using Smartwatches. *Scientific Reports*, 9(1):15054, 2019.

[46] Q. Yousef, M. B.I. Reaz, and M. A.M. Ali. The analysis of PPG morphology: Investigating the effects of aging on arterial compliance. *Measurement Science Review*, 12(6):266–271, 2012.

[47] Allison A. Merz and Susan Cheng. Sex differences in cardiovascular ageing. *Heart*, 102(11):825–831, 2016.

[48] Leandro Pecchia, Rossana Castaldo, Luis Montesinos, and Paolo Melillo. Are ultra-short heart rate variability features good surrogates of short-term ones? State-of-the-art review and recommendations. *Healthcare Technology Letters*, 5(3):94–100, 2018.