UNIVERSIDAD DE GRANADA

Bachelor's Thesis

Bachelor's Degree in Computer Science

Hyperparameter optimization in Deep Neural Networks

In the context of EEG classification

Author

JAVIER LEÓN PALOMARES

Supervisors

JULIO ORTEGA LOPERA

SAMUEL ROMERO GARCÍA

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, June 2018

JAVIER LEÓN PALOMARES:
Hyperparameter optimization in Deep Neural Networks
In the context of EEG classification.

Supervised by:
JULIO ORTEGA LOPERA
SAMUEL ROMERO GARCÍA

Granada, June 2018

It is good to have an end to journey toward; but it is the journey that matters, in the end.

— Ursula K. Le Guin

RESUMEN

Gracias a los avances de los últimos años en potencia de cómputo, el campo de las redes neuronales artificiales ha tenido un importante resurgimiento que ha traído grandes progresos en muchos problemas que se creían demasiado difíciles. Sin embargo, esta potencia renovada trae consigo un incremento en la dificultad de encontrar la configuración más apropiada para cada problema en concreto.

En este trabajo se propone un método evolutivo incremental para optimizar hiperparámetros en redes neuronales. Tras una fase previa de selección de características, se hace evolucionar una primera población de redes neuronales para encontrar arquitecturas que proporcionen un rendimiento mayor al inicial. Después, en una segunda fase se hace evolucionar otra población con el objetivo de encontrar la mejor combinación de ratio de aprendizaje, tasa de *dropout* y épocas de entrenamiento. Adicionalmente, en cada una de estas fases es posible aprovechar o bien el paralelismo implícito de una GPU o bien el paralelismo a nivel de distribución de tareas a distintas CPUs.

En su aplicación a clasificación en visualización motora usando interfaces cerebro-máquina, este método parece conseguir resultados muy prometedores en cuanto a precisión, usando tanto redes neuronales como clasificadores más simples como SVM.

PALABRAS CLAVE: Interfaces cerebro-máquina (BCI) · Algoritmos genéticos · Redes neuronales artificiales · Selección de características · Optimización de hiperparámetros

ABSTRACT

With the increase in computational power in the last years, neural networks have seen a grand revival that has brought along significant progress in many problems that were once considered too hard. However, this newfound power comes hand in hand with a raise in the difficulty of finding the optimal configuration for a given task.

In this work we propose an incremental evolutionary method for optimizing hyperparameters in neural networks. Starting with a feature selection phase, a population of neural networks is first evolved to find appropriate hidden layer configurations (*structure optimization*); then, another population is evolved in order to find the best combination of learning rate, dropout rate and number of epochs (*learning optimization*). Every one of these three steps can leverage implicit GPU parallelism and explicit CPU task distribution if the computational loads demand it.

Applied to classification of motor imagery tasks for brain-computer interfacing (*BCI*), this method appears to produce very promising results in terms of accuracy, both using neural networks and some simpler classifiers like Support Vector Machines.

KEYWORDS: Brain-computer interfaces (BCI) · Genetic algorithms · Artificial neural networks · Feature selection · Hyperparameter optimization

ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisors, Julio and Samuel, for giving me the opportunity to work in a project I have enjoyed at a personal level.

I would also like to highlight the work of those professors whose unfaltering dedication has helped many students like me become a bit more of an engineer.

A special mention is owed to the University of Essex, and to John Q. Gan in particular, for providing the dataset used in this work.

Finally—for how could it have been otherwise—, I wish to acknowledge the support of my family and friends, who ultimately make life worth living.

Many thanks, especially, to Eva,
and to Laura, Germán, Cristina, Antonio and Sergio,
for helping me get back on the right track.

This project has been partially funded by grant TIN2015-67020-P (Spanish *Ministerio de Economía y Competitividad* and *European Regional Development Fund*).

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACRONYMS

BCI	Brain-Computer Interface
EEG	Electroencephalography
ECoG	Electrocorticography
ANN	Artificial Neural Networks
EA	Evolutionary Algorithms
ReLU	Rectifier Linear Unit
ELU	Exponential Linear Unit
MRA	Multiresolution Analysis
DWT	Discrete Wavelet Transform
MI	Motor Imagery
NSGA-II	Nondominated Sorting Genetic Algorithm II
SPEA2	Strength Pareto Evolutionary Algorithm 2
PAES	Pareto Archived Evolution Strategy
SVM	Support Vector Machine

Part I

INTRODUCTION

"What do all these fancy words mean?"

1 CONTEXT

In the famous movie *The Matrix*, humans are plugged into the evil matrix through a port in the back of their skulls. In the 1995 Japanese animated movie *Ghost in the Shell*, a number of characters also sport a similar device (this time voluntarily) in order to perform several tasks, a remarkable one being diving into virtual reality.

While there is no telling when that level of technology will become everyday for us, numerous promising advances already point to its realization. This work aims to study a part of the state-of-the-art in this respect: machines learning to understand us.

In this first chapter we provide a brief context. In sections 1.1 and 1.2 we talk about two of the closest related fields. The main techniques we will be using are introduced in sections 1.3 and 1.4. In subsequent chapters, the whole mechanism will be pieced together to finally reach some conclusions.

1.1 BRAIN-COMPUTER INTERFACES

Many variations of the contraptions we mentioned at the start fall into the term Brain-Computer Interface (BCI). BCIs function as communication channels that do not rely on peripheral nerves or muscles [1]; put in a practical way, they can allow for a compensation or even augmentation of human abilities.

While not as futuristic as in fiction, the near future prospects of BCI technology range from convenient to humanitarian—from brain-operated communication channels [2] to medical diagnosis or advanced systems which aid paralyzed patients in their daily lives [1].

1.2 ELECTROENCEPHALOGRAPHY

As the name Brain-Computer Interface suggests, we need a tool in order to link our brain to a computer. It can be of one of three kinds:

- Invasive: these include the most powerful applications, like vision and mobility restoration; brain implants are an example of this. Nonetheless, they come with important downsides, such as neurosurgery and its collateral effects.

1 A chip in your brain can control a robotic arm. Welcome to BrainGate. [Accessed 29-05-2018]

- Non-invasive: when surgery is not appropriate, there exist other technologies that trade in precision for ease of use and lower costs. Among these we find Electroencephalography (EEG).
- Partially invasive: a compromise between the two in the form of a not-so-aggressive procedure that only reaches the outside of the brain. Electrocorticography (ECoG) is a promising method.

In particular, the dataset employed in the experiments of this document was obtained with EEG. This paradigm utilizes a set of electrodes placed on the scalp to record electrical activity from the brain; those readings are later useful for medical diagnosis or diverse engineering purposes (ours being one of them).

1.3 ARTIFICIAL NEURAL NETWORKS

Artificial Neural Networks (ANN) have existed for many decades as a theoretical model. However, it was not until a few years ago, with the significant increases in computational power, that we started to grasp some of their true potential.

A classic definition encompassing the main points could be the following:

> *A neural network is an interconnected assembly of simple processing elements, units or nodes, whose functionality is loosely based on the animal neuron. The processing ability of the network is stored in the interunit connection strengths, or weights, obtained by a process of adaptation to, or learning from, a set of training patterns. [3]*

As far as we are concerned, this means that ANNs draw inspiration from nature to build a model capable of prediction from inputs, given sufficient training. And, as we will see in subsequent chapters, the key is precisely the training phase. It is the most arduous task in order to get desired outcomes, for it requires tuning a number of parameters.

1.4 EVOLUTIONARY ALGORITHMS

Again influenced by nature, Evolutionary Algorithms (EA) try to mimic the phenomenon that takes place in communities over extended periods of time: with the passing of generations, better adapted individuals prevail over less fit ones; this process makes for an intuitive—and highly effective—optimization technique.

EAs enjoy widespread popularity in many fields due to them not making structural assumptions about the problem in question, along with their reasonable time performance. Owing to these reasons, we will look into them as our main optimization mechanism.

1.5 PUTTING EVERYTHING TOGETHER

The matter at hand is one of classification of samples into one of three categories. The samples consist of data extracted using EEG, a kind of BCI, that represent an imagined movement of certain limbs. We will see this in chapter 4.

Once we have the data, the next stage is the classification. This is achieved first through a feature selection phase (chapter 5); next, we will explore a few forms of optimization for neural networks (described in chapter 6). After performing several experiments, we will gather the information and draw some conclusions on how to build the best model according to our capabilities and resources.

The last touch would be to integrate hardware and software into a functioning product, which is of course easier said than done. That task is outside the scope of this academic setting.

Let us now proceed to explain in chapter 2 a variety of general concepts that are required to fully understand the technical side of this work.

2

BACKGROUND

2.1 NEURAL NETWORKS AS CLASSIFIERS

One of the most popular use cases of ANNs is supervised learning; what distinguishes it from unsupervised learning is the existence of labeled examples that guide the learning stage, against the need to infer all information. Supervised learning can be further divided into classification and regression: the former aims to split a set of data into two or more groups, while the latter is more of a function approximation. Since our job will consist in recognizing several types of EEG patterns, the discussion will focus on the particular details of classification from here on.

A neural network, in its most basic form, is made of simple processing units (called *neurons*) distributed among three kinds of layers:

- Input layer: with as many neurons as characteristics describing a single sample. Each unit receives the original value of its corresponding characteristic. Depending on the problem (and not just for this sort of model), it may be useful to preprocess the data so as not to bias the model.
- Output layer: the number of possible outputs dictates its number of neurons. Only one of them can be active at a time, indicating the answer of the neural network to the classification question.
- Hidden layers: these are inserted between the previous two. Their quantity ranges from one to hundreds in the most extreme applications, and their interactions yield the predictive ability of the model.

A typical vanilla feed-forward neural network contains directed connections between its different layers, but not between components of the same layer. Also, the term *feed-forward* implies that there are no cycles in the graph of the network, and so the information moves from the input to the output through the hidden layers. The way in which this information flows is defined by the *activation function* (the response of a neuron from an input number) as well as the *weights* of the connections between neurons: although it is frequent to have fully-connected layers, it does not mean that one unit passes the same value to all the other units it points to, or even that they are not null; finding it out is precisely the task of the learning algorithm.

To get a better picture of the concept, the following figure illustrates a generic model with one hidden layer:

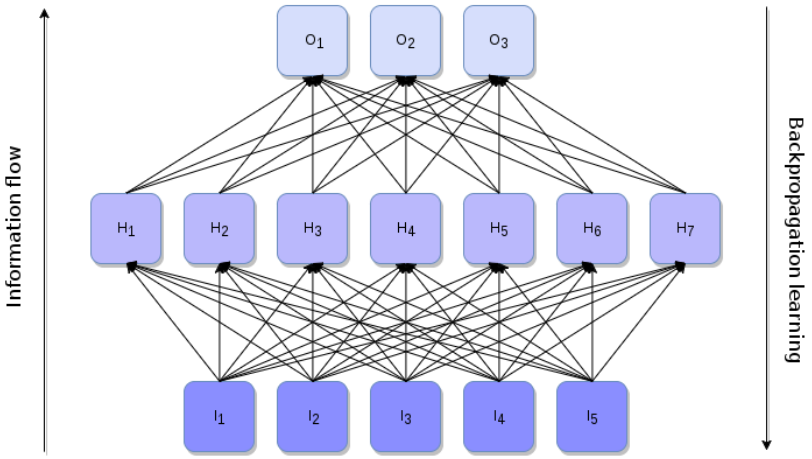

Figure 2.1: A neural network with one hidden layer.

We can see that the hidden layer has its neurons labeled H_i. The input layer is represented by the I_i units and the output layer corresponds to the different O_i.

This image introduces another new concept: *backpropagation*. Whereas the information is transformed by means of the weights and transmitted forward, the adjustment of those weights also needs of a propagation in the opposite direction. To sum it up, backpropagation has two main steps:

1. Propagation. Generate predictions for the training examples, then calculate the error at the output layer; a common error measure is the squared difference between the actual value and the expected value. Afterwards, recursively propagate the error calculations to the successive hidden layers taking into acount the already computed error values, until the input layer is reached.

2. Weight update. Multiply the value of the activation function of each neuron and its error obtained in the first step. This is called the *gradient* in *Gradient Descent*. Finally, subtract a fraction of this gradient from the weight; that fraction (*learning rate*) has a significant effect on the process, for a value too high can cause the algorithm to jump over a local minimum without reaching it, and a value too low can overextend the training time.

The above procedure is repeated until the model's performance is adequate. If we wish to speed it up with a reasonable tradeoff in quality, we can use not the whole training set for each iteration but smaller subsets called *batches* (*Stochastic Gradient Descent*). This allows us to maintain enough generalization while at the same time reducing the cost of gradient computation [4].

A problem related to backpropagation is the *vanishing gradient*, which only occurs in the training stage of neural networks. Because of the chain rule used to propagate the error through the layers, if the activation function has a near-zero derivative at some points, the gradient will approach that value too. In worst-case scenarios, it results in a practical standstill of the training.

Multiple solutions have been proposed in the past few years, and research on the topic is still ongoing. A few popular activation functions as of now include:

- *Hyperbolic tangent*: the classic, due to it being bounded, which increases the efficiency of the training. However, its shape at both ends produces the vanishingly small gradients.
- *Rectifier Linear Unit (ReLU)*: defined by $f(x) = 0$ if $x \leq 0$ and $f(x) = x$ if $x > 0$. It avoids small gradients for high input values and makes networks sparser with its negative part (accounting again for more efficiency). The downside is the *dying* ReLU, in which some neurons become perpetually inactive.
- *Leaky ReLU*: one of the variants of ReLU which tries to avoid inactive neurons by setting $f(x)$ for negative x to $0.01x$ instead of 0. Its generalization is the *Parameterized ReLU* [5, 6].
- *Exponential Linear Unit (ELU)* [7]: another recent alternative that replaces the horizontal section of the ReLU with an exponential function.

These functions can be observed in the following figure:

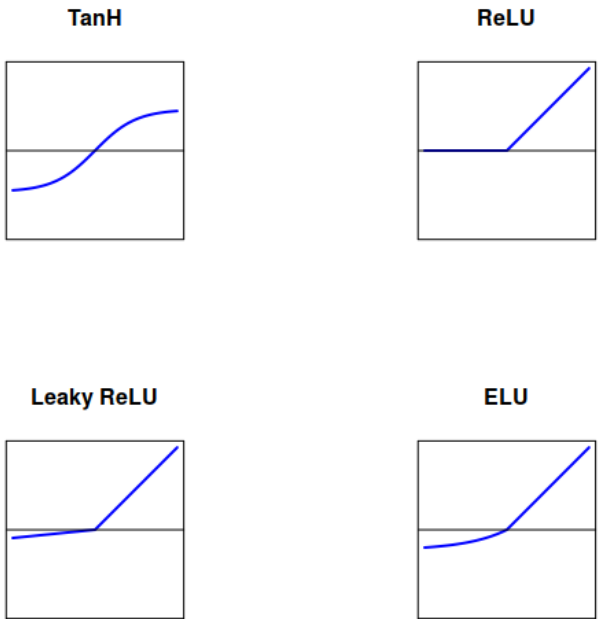

Figure 2.2: Four of the most popular activation functions.

In general, as the complexity of a model grows, so does the difficulty of its training. Neural networks are, in theory, capable of approximating every possible function, and this comes at a price: some of the following common issues in machine learning [8] have increased consequences.

- Limited data: without delving too much into the underlying theory, the more powerful a model is, the more training samples are needed in a substantial proportion. Coupled with it, the smaller the training set, the smaller the chance it will represent the whole population.
- Imbalanced data: when there are far more examples of one or more classes than of the rest, we speak of imbalance. Apart from not representing the population, this phenomenon can even bias a classifier in a way that it only outputs one class, as it seems the best way to minimize the error.
- Incomplete data: this is the case when the available samples have missing values. In this situation there are two options: either we discard incomplete fields from across all samples or we try to infer them.
- High dimensionality: linking with the first bullet point, powerful models can suffer of *overfitting* with limited and/or high-dimensional samples. The latter can cause the model to just learn the available data "by heart" and fail to generalize to new examples. Even leaving extreme cases aside, the number of features has a direct impact on the time required for training.

It is clear that we must minimize the hurdles posed by the data, and also make sure we pick the right parameters. For this second task, we have Evolutionary Algorithms (EA).

2.2 EVOLUTIONARY ALGORITHMS

Evolutionary algorithms are iterative procedures that take a population of feasible solutions to a problem and try to progressively find the best possible answer by mimicking nature in its selection process.

Being general-purpose metaheuristics, they find use in fields as disparate as computer science, medicine, economics or cryptography, among many others. In the business at hand, they will serve as neural network optimizers, in addition to feature selectors in a preliminar step.

The kind of EAs we will be using are known as *genetic algorithms*. Now, if we analyze them, we find they have several main pieces:

- Individual: a potential solution to a problem, represented in an adequate manner.
- Population: a set of individuals for a given iteration (*generation*).
- Fitness: every genetic algorithm has one or more fitness functions which evaluate the quality of an individual. It can be the most expensive operation, depending on the situation.
- Selection process: the creation of new individuals (*offspring*) needs a set of parents. The selection process chooses them based on some criteria, usually involving the fitness.
- Crossover operator: once we have a list of potential parents, this operator defines how to create *offspring* that resemble them in some way.
- Mutation operator: in order to maintain diversity, random mutations are introduced with a certain probability.
- Replacement: how the old population and the offspring set will be used to create a new population.

Strong emphasis is placed on a solid balance between finding novel solutions and improving the best ones yet, also known as the *exploration-exploitation tradeoff*. Hence, all the pieces as a whole must attain a delicate synergy. With that in mind, let us go into a bit more detail in the next sections.

2.2.1 *Individuals*

The individuals of a genetic algorithm may be represented in a variety of ways. It is crucial to find the best suited for our needs, as it will not only impact time and space efficiency but also the overall performance of the optimization, enabling some genetic operators and discarding others.

For example, in feature selection we could code our solutions as a binary vector in which the ones represent active features and the zeros inactive features. Alternatively, we could just keep a list of the indices of active features; this would use less memory but also make some very simple operators in binary codification way more convoluted to implement.

Common representations include binary, order-based and real-valued. Having already exemplified the first, order-based is natural in *Travelling Salesman Problem*-like cases and real-valued can be found in numerical function optimization.

2.2.2 *Population*

Population size must have an equilibrium between enough individuals to explore the solution space and not too many of them, which would lead to excessive running times.

Another relevant discussion is how to initialize the first set of individuals. For the sake of a good exploration, it should include a decent portion of the search space, often taking advantage of randomness.

2.2.3 *Fitness*

Fitness measures, both in type and in number, depend on what aspect of a problem we want to tackle.

If we hope to find a global maximum or minimum in a complex function, the fitness will just compute the value at a given point. If we want to choose from a set of features, the evaluation will need to assess how good a subset is by training and testing a machine-learning model; similarly, if we wanted to further optimize that model, we would need to try many variations of its parameters.

However, suppose we can't decide on what we prefer to improve: for instance, we want a solution to be as accurate as possible, but we want it to be simple, too. This is when *multiobjective optimization* comes in: instead of returning the best candidate, we accept a compromise between the two criteria and return a list of candidates that are not better or worse than the others. They just score well in both measures but in different proportions.

2.2.4 *Selection*

Parent selection process is one of the keys for the success or failure of the algorithm. Since it is a problem-independent part of the algorithm, in contrast with the more specific crossover and mutation operators, we can reduce our choices to a handful of classic techniques [9]:

- Proportional: the probability of picking an individual is driven by its relative fitness with respect to the overall sum.
- Tournament: n random individuals are set side by side and compared by fitness score, after which the best is taken. Repeat until the parent quota is met.
- Ranking (any kind): with the population sorted by fitness, it assigns order-based probabilities following some function. The catch here is that the probabilities are not directly related to the measured quality of the solutions, which allows for more control over convergence-exploration.
- Randomized: little to no influence of fitness in the decision.

2.2.5 *Crossover*

There exist a plethora of options for this highly representation-dependent operator. As it is unfeasible to list them all, even just the most relevant ones, we will limit this section to the kinds we will use. The only prerequesite is that they offer a degree of continuity from parent to offspring.

2.2.5.1 *In binary representation*

Like we mentioned earlier, a binary representation consists of a series of zeros and ones. While not the most memory-efficient alternative, it results in fairly straightforward yet powerful operators:

- Single-point: choose a random position between the first and the last. To each of its sides we only add the elements of a single parent.

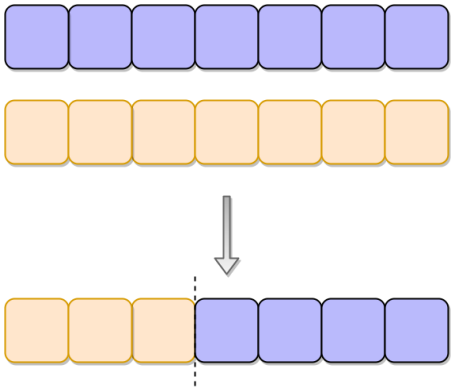

Figure 2.3: Single-point crossover example.

- Double-point: this time, generate two random positions. Use them to alternate the two parents' elements. Note that this idea can be extended to n-point crossovers.

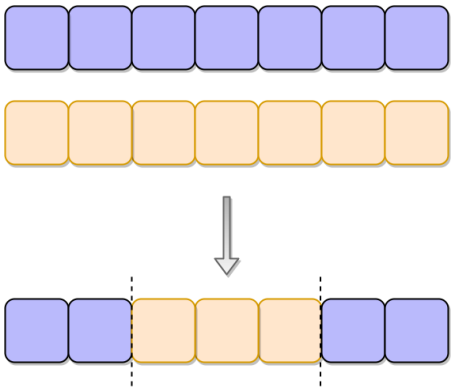

Figure 2.4: Double-point crossover example.

- Uniform (UX): choose a uniform real number between 0 and 1 and use it to decide which of the two parents the *ith* bit will come from. From a practical standpoint, this means that bits shared by both of them will always be passed on to the next generation.

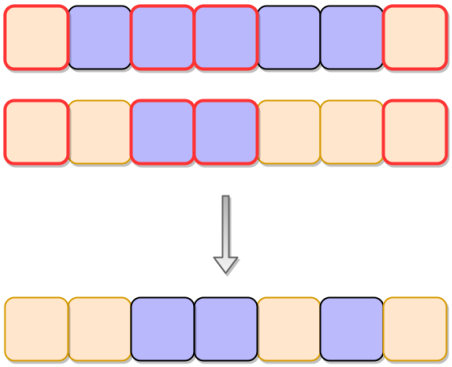

Figure 2.5: Uniform crossover example. Note that the highlighted elements are present in both individuals.

2.2.5.2 *In neural networks*

If binary representations and crossovers are among the most widely studied, it is not quite the case with neural networks in genetic algorithms (probably because the revival of the field happened not many years ago).

Some parameters (*hyperparameters*, because their values are fixed before the training phase) to tune are the number of hidden layers, or neurons per layer, or the learning rate.

For the first two, one could do something along the lines of the binary crossovers we just explained: take some layer sizes from one parent and the rest from the other, have the child get as many hidden layers as the average of its parents, etc. Perhaps we want to keep the numbers inside tolerance intervals, so we should check after each operation too.

To pass learning rates on, maybe a simple average would do. Given that we don't have prior knowledge about the function we are trying to optimize—only that it has a global optimum somewhere—, it would be a reasonable strategy: if the optimum is between two values, an average would keep us in range; if not, we would not have known anyway.

2.2.6 *Mutation*

Evolution needs some diversity, and what we have not done with selection and crossover, we must do with mutations in the offspring. Again, the mutation operator can take many forms, so we will restrict ourselves to what is applicable.

2.2.6.1 *In binary representation*

The easiest way to mutate a vector of binary elements is to randomly flip one or more of them i.e. write 1 where there was 0 and viceversa. Both the amount of flips and the probability of a mutation rule how much diversity is introduced.

2.2.6.2 *In neural networks*

When dealing with quantities, such as the number of layers or of neurons per layer, the simplest thing to do is to add or subtract. A more frequent mutation could be to just alter the number of neurons in one (or more than one) layer either by a fixed amount or by using a probability distribution (gaussian noise, for instance). A much rarer mutation—for its impact in the structure of the network—could involve a slight change in the number of layers.

2.2.7 *Replacement*

The act of combining the old population and the freshly-created offspring into a new population for the next generations has a great significance in the final outcome of the algorithm. Let us start reviewing the two categories of genetic algorithms according to the magnitude of the replacement:

- Steady-state: only a few parents—typically two—are selected for reproduction in each generation, and the offspring they yield replace the same amount of individuals in the original population.
- Generational: the whole previous population is displaced by a newly created group.

Since these categories are not strict, we could have a genetic algorithm that ranks halfway between the two: not all the previous individuals have to be replaced. We usually make this decision with the help of additional criteria: imagine that we lump together all the individuals and sort them by fitness; all offspring could be better than the best old individual, and thus the entire population would be replaced, but it might very well not be the case, and end up with a mix of old and new.

The way in which we choose to do the replacement causes, along with the selection procedure, a certain degree of *selective pressure* [9]. Selective pressure characterizes the extent to which better individuals are prioritized in the evolution stages of the algorithm.

Finally, here is a graphic overview of how a genetic algorithm works:

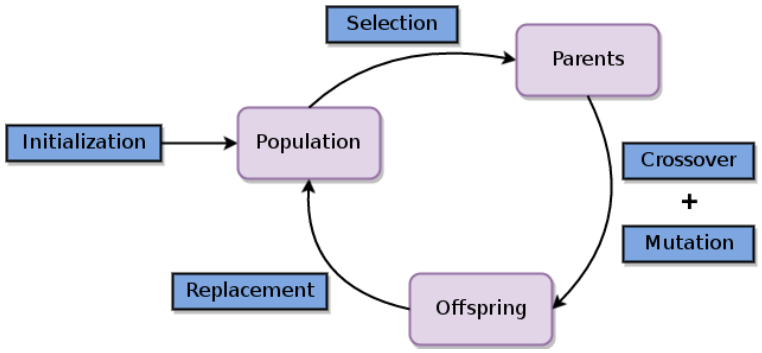

Figure 2.6: General overview of a cycle in a genetic algorithm. Blue boxes represent operations. Violet boxes represent sets of individuals.

It is important to note that this basic scheme is complemented by the stopping conditions, which range from finding the optimum (when we have the number but need the associated solution) to an artificial limit by iteration count or a streak of non-productive generations.

2.3 MULTIOBJECTIVE OPTIMIZATION

As was briefly discussed in a previous section, it is possible to have several criteria to guide our search of good solutions for our problem.

We can define multiobjective optimization in a more formal way:

$$\min_{x \in X}(f_1(x), f_2(x), ..., f_n(x)) \tag{2.1}$$

Where X represents the space of possible solutions, and $n > 1$ is the number of objective functions. An alternative notation would be taking $F(x)$ as a vectorial function containing all $f_n(x)$.

When looking for suitable solutions, we pay special attention to *Pareto-optimal* solutions, which are *non-dominated*. Domination of x_1 over x_2, with $x_1, x_2 \in X$, can be defined by the following two conditions:

1. $f_i(x_1) \leq f_i(x_2), \forall i : 1 \leq i \leq n$
2. $f_i(x_1) < f_i(x_2)$, for one or more $i \in \{1, 2, ..., n\}$

The *Pareto front* is the set of solutions which meet both conditions for the rest of solutions but fail to meet the first one when compared among themselves. An example of a Pareto front is shown in the next figure:

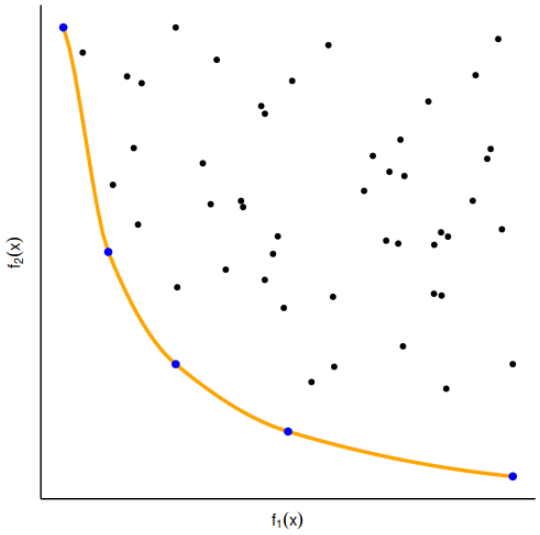

Figure 2.7: A two-dimensional Pareto front with its points highlighted in blue and an approximate curve outlined in orange.

A multiobjective optimization algorithm typically returns the Pareto front in its entirety, and it is our duty to make an informed decision about which element(s) we deem more appropriate.

2.4 FEATURE SELECTION

When we face a complex classification or regression job, we often have a lot of unorganized information. In these cases, one of the keys for success is to weed out whatever data that does not contribute anything useful. An excess of data does not just increase the computational cost of training our models, it can also hamper its performance—and in no small proportion, to make things worse.

It is therefore clear that we need some kind of method for choosing the right information to keep. However, achieving this is no easy feat, as it falls into the category of *NP-hard* problems [10]. For this reason, a number of approximate alternatives have been proposed. They can be of one of three classes [11].

Wrappers use the performance of a model to score different feature subsets. *Filters* select features as a preprocessing step, independent from any models we later use. *Embedded* methods incorporate the selection in their training processes. Wrappers tend to yield the best outcomes because of their more extensive search, and are a good option if given enough computational resources; with these conditions met, we will be using genetic algorithms, a popular example.

As we want to have more than one measure of classification quality, we will use a multiobjective genetic algorithm. Picking up on what we said in the last section, Pareto-based approaches are quite common. Among these, the state of the art is mainly comprised of some of the most famous algorithms: Nondominated Sorting Genetic Algorithm II (NSGA-II) [12], Strength Pareto Evolutionary Algorithm 2 (SPEA2) [13] and Pareto Archived Evolution Strategy (PAES) [14].

The first one is the most widely used by a large margin, and we will use it too for two main reasons: its higher general performance against the other alternatives, along with its better population diversity (refer to the original paper for more details), is a decisive factor; in addition, since it works with a single population instead of keeping a concurrent one for the best individuals, it is easier to implement and conceptually more intuitive.

One important downside of multiobjective genetic algorithms is, though, that they do not perform so well with many objectives. But, since we will use a maximum of two or three, it should not become a problem for us.

Having now reached this point, we have refreshed our knowledge in machine learning, neural networks, evolutionary algorithms and feature selection, all of which will prove useful later on. Following chapter 3, which only concerns the objectives of this project, we will start going in-depth into the matter in part II.

3

MOTIVATION AND OBJECTIVES

3.1 MOTIVATION

In recent years, the field of machine learning has seen substantial advances. Due to the emergence of a wide array of novel techniques and the significant increase in computational power, state-of-the-art models outperform older methods in many applications. Deep Neural Networks are the most successful example, achieving results comparable to those of humans in tasks such as image recognition or game playing.

Because of this, a certain degree of knowledge or familiarity with machine learning is a valuable asset for any Computer Science or Computer Engineering graduate. Thus, this work aims to serve as an opportunity to acquire more insight into the capabilities of neural networks in conjunction with related optimization methods. Additionally, since we will be using real data instead of typical testing datasets, it will provide some experience about tangible problems and their complexity.

My personal motivation is not too far from the above. As a sci-fi reader, the idea of machines solving tasks better than humans is always present. True machine intelligence may not come soon, if it ever does, since there is a long road ahead. However, experience demonstrates that we are too early to either dismiss or take for granted, and so the only way to know is to try, step by step. Although not as fancy as what can be found in *Hugo* and *Nebula* award winners, machine learning is already a reality in terms of usefulness, and that is what I am interested in.

3.2 OBJECTIVES

The objectives of this work are a natural consequence of everything stated up to this point. We can condense them as follows:

- Gain practical expertise on the use of neural networks (in particular, training parameters and optimal structures).
- Learn about different optimization techniques and algorithms that can complement neural networks. We are mainly looking at genetic algorithms.
- If possible, build a general enough codebase which can be reused and extended for further experiments.

Part II

METHODOLOGY

4 THE DATASET

In order to proceed with our machine learning endeavors, we need a dataset to experiment on. This chapter describes it, along with a common issue in many datasets that happens to arise here too—and which is the motivation for chapter 5.

4.1 DESCRIPTION

As stated in Chapter 1, we will be working with EEG readings. The patterns have been built using a kind of Multiresolution Analysis (MRA) [15] called the Discrete Wavelet Transform (DWT), applied in [16] to characterize EEGs from Motor Imagery (MI) tasks. On this occasion, our aim is to use them to distinguish between an imagined movement of the left hand, the right hand or the feet.

The feature extraction procedure yields a variable but always huge number of coefficients. This number is determined by:

- S: number of segments.
- E: number of electrodes.
- L: number of levels.

If we input them in the formula $2 \times S \times E \times L$, we obtain how many sets of coefficients describe the pattern. In turn, each set can contain from 4 to 128 coefficients. For the experiment at hand, $S = 20$, $E = 15$ and $L = 6$, making a total of 3600 sets and the overall limit being 151200 features.

Again, [16] proposed a way of reducing this amount by computing the variance of each set, leaving us with 3600 features altogether.

Whereas this is a massive reduction in dimensionality, the scarcity of sample patterns (about 360 in total) still poses a challenge for the task of classification at hand. We will try to squeeze out as much performance as we are able from this version of the dataset, but we must not forget that we can still work with the full version in other ways.

4.2 THE CURSE OF DIMENSIONALITY

When dealing with high-dimensional spaces (those with hundreds or thousands of dimensions), we face issues that are often not present in simpler ones. As the volume of the space grows, the samples become sparser and thus they fail to represent the whole set of possibilities in a statistically significant way.

Applied to machine learning, the number of features plays a pivotal role: a low count may not provide enough information, and a high count may prevent a good generalization by introducing superfluous or noisy details. Thus, the key lies somewhere in the middle: we have to find a delicate balance so that our models do not *underfit* nor *overfit*.

The latter is our main concern when trying to bring down the feature count, so we will restrict ourselves to a subset smaller than the training one. Since we are going to split the (approximately) 360 samples evenly for training and testing, this means that we should consider at most 180 features—ideally, only a fraction of that.

As a final note, it is important to mention that we have several instances of this dataset structure that correspond to different test subjects (namely, 104, 107 and 110). These subjects were the best performers in the recordings, because one of the downsides of EEG and MI is that the person has to learn how to use the device. The other subjects were not so good at the task, and so they will not be considered here.

Wrapping up this chapter, we have gone over how the dataset is constructed and how an important hurdle, the curse of dimensionality, springs from it. The very next chapter is dedicated to overcome said obstacle, and from then on we will build a solution for what was our main goal—classification.

5 FEATURE SELECTION

Like we mentioned before, the chosen approach for the feature selection step is a genetic algorithm. With a few modifications specific to our problem, the basic structure will be that of NSGA-II.

In chapter 2 we hinted at a binary representation to solve this task. It not only facilitates the overall implementation, but—as we already said—it also brings with it some easy yet effective operators.

At the end of this chapter we will have the knowledge to take on feature selection for cases where the total amount of predictors is not absurdly high. Every relevant operation will be explained so that equivalent outcomes can be achieved.

5.1 FEATURE SELECTION PROCEDURE

The main body of the algorithm corresponds to a typical NSGA-II layout. Let us see its general form before going over the different parts:

Algorithm 1: NSGA-II

```
Procedure NSGA-II
    Input: population size, generations, data, max features
    Output: final population
    population ⟵ Initialize(population size, max features);
    evaluation ⟵ Evaluate(population, data);
    population ⟵ NDSort(population, evaluation);
    for gen = 0 to max generations do
        parents ⟵ Selection(population);
        offspring ⟵ CreateOffspring(parents);
        shared population ⟵ population ∪ offspring;
        evaluation ⟵ Evaluate(shared population, data);
        shared population ⟵ NDSort(shared population,
        evaluation);
        population ⟵ Replace(shared population, population
        size);
    end
    return population;
end
```

Notice that it returns the whole population from the last generation. Ideally, we want to take its first Pareto front and choose whatever solution we deem more appropriate for our needs.

We can proceed to go now into more detail about the different functions that make up the algorithm. At the beginning, there is a randomized initialization of the population, so as to have something to start with:

Algorithm 2: Population initialization

```
Function Initialize
    Input: population size, max features
    Output: population
    population ⟵ ∅;
    for i = 0 to population size do
        population ⟵ population ∪ RandomVector(max
        features);
    end
    return population;
end
```

In the actual code, each element of the population is created as a sequence of zeros which is then modified to introduce ones in random positions. These two numbers correspond to the intuitive notion of boolean false and true, respectively, telling us whether a given feature is chosen or not.

After that, and also every time we create a new population, we have to evaluate the fitness of its individuals. This is accomplished in `Evaluate`:

Algorithm 3: Population evaluation

```
Function Evaluate
    Input: population, data
    Output: evaluation
    evaluation ⟵ EmptyMatrix(population size, objective
    count);
    for obj = 0 to objective count do
        for ind = 0 to population size do
            evaluation[ind][obj] ⟵ F_obj(population[ind], data);
        end
    end
    return evaluation;
end
```

Here we assume that we already have the different F_i at our disposal. Also, not all fitness functions necessarily use the data for their computations—we will discuss this at the end of the chapter—but it is written this way for uniformity.

The next operation is one of the keys of NSGA-II: the non-dominated sorting. Let us first see the reasoning behind it before explaining how it is carried out.

Diversity of solutions in the Pareto fronts is one main issue that NSGA-II tries to address. It is clear that pertaining to a better Pareto front makes an individual rank higher; however, when comparing two individuals from the same front, we may want to choose that which is most distant to its neighbors. This is because similar solutions tend to score similarly in all aspects, but we need them to be as different as possible in order to explore the space better.

This way, individuals with significant quality will naturally be in the top positions, but at the same time we make sure that differences are favored too. For clarity, we will break down the process into two steps, explained with the help of the original paper ([12]). Also, we will use some auxiliar functions to abstract behavior not relevant to our understanding of the concept.

Algorithm 4: Non-dominated sort steps

```
Function NDSort
    Input: population, evaluation
    Output: population
    fronts ⟵ ComputeFronts(evaluation);
    fronts ⟵ ComputeDistances(fronts, evaluation);
    population ⟵ ∅;
    for F_i in fronts do
        population ⟵ population ∪ F_i;
    end
    return population;
end
```

The front computation step assigns a front to each individual of the population based on its fitness scores. Before delving into the details, let us define a few things:

- $p \prec q$: "p is not worse than q in any objective and is better in at least one", for two individuals p and q. It is read as "p dominates q".
- S_p: the set of individuals dominated by p.
- n_p: the count of individuals that dominate p.
- F_i: the different fronts. A lower i means a better overall quality of the elements.

Remember that `evaluation` contains a row for every individual and that its elements are the different fitness scores; this means that we can think of an individual in terms of its associated row in this structure.

The pseudocode is as follows:

Algorithm 5: Front computation

```
Function ComputeFronts
    Input: evaluation
    Output: fronts
    fronts ⟵ {F_1, F_2, ..., F_n} ;        /* All of them empty */
    for p ∈ evaluation do
        S_p ⟵ ∅;
        n_p = 0;
        for q ∈ evaluation do
            if p ≺ q then
                S_p ⟵ S_p ∪ {q};
            end
            else if q ≺ p then
                n_p = n_p + 1;
            end
        end
        if n_p is 0 then
            p.front = 1;
            F_1 ⟵ F_1 ∪ {p};
        end
    end
    i = 1;
    while F_i ≠ ∅ do
        Q ⟵ ∅ ;          /* Members of the next front */
        for p ∈ F_i do
            for q ∈ S_p do
                n_q = n_q - 1;
                if n_q is 0 then
                    q.front = i + 1;
                    Q ⟵ Q ∪ {q};
                end
            end
        end
        i = i + 1;
        F_i ⟵ Q;
    end
    return fronts;
end
```

The second step (Algorithm 6) will tell us how far each individual is from its neighbors—within its own front—on average.

Algorithm 6: Crowding distance computation

```
Function ComputeDistances
    Input: fronts, evaluation
    Output: fronts
    for F_i in fronts do
        n = size(F_i);
        for j = 0 to n do
            F_ij.distance = 0;
        end
        evaluation_i ⟵ GetEvaluations(evaluation, F_i);
        for obj in columns(evaluation_i) do
            objective ⟵ Sort(evaluation_i[...][obj]);
            F_i ⟵ SortBy(F_i, objective);
            F_i1.distance = F_in.distance = ∞;
            range = objective[n] - objective[1];
            for k = 2 to n − 1 do
                F_ik.distance = F_ik.distance +
                (objective[k+1]-objective[k-1]) / range;
            end
        end
        F_i ⟵ SortByDistance(F_i) ;     /* Descending order */
    end
    return fronts;
end
```

We are now able to upgrade our $\prec$ operator to take into consideration the crowding distances. Let $p \prec_{nds} q$ be true if p.rank $<$ q.rank or p.rank $=$ q.rank and p.distance $>$ q.distance. We can use it now in our parent selection process:

Algorithm 7: Selection operator

```
Function Selection
    Input: population
    Output: parents
    parents ⟵ ∅;
    for i = 0 to size(population)/2 do
        p, q = RandomChoice(population, 2);
        if p ≺_nds q then
            parents ⟵ parents ∪ {p};
        end
        else if q ≺_nds p then
            parents ⟵ parents ∪ {q};
        end
    end
    return parents;
end
```

There are several ways of deciding which individuals from the last generation we are going to use to create offspring. One of the most common is the *Binary Tournament Selection* described here. The size of the parent pool is fixed to half the population in this instance, but it can be whatever we think is suitable.

Our next function is in charge of making the offspring:

Algorithm 8: Offspring creation

```
Function CreateOffspring
    Input: parents
    Output: offspring
    offspring ⟵ ∅;
    for i = 0 to size(parents) do
        p1, p2 = RandomChoice(parents, 2);
        if Random() ≤ crossover probability then
            child ⟵ Crossover(p1,p2);
            if Random() ≤ mutation probability then
                child ⟵ Mutation(child);
            end
            offspring ⟵ offspring ∪ {child};
        end
    end
    return offspring
end
```

Some parameters are up to the designer, such as how many offspring are returned, the crossover and mutation probabilities, or the operators. For the latter, some common alternatives are found in Algorithms 9, 10, 11 and 12.

Algorithm 9: Single-point crossover

```
Function SinglePoint
    Input: p1, p2
    Output: child
    child ⟵ p1;
    pivot = RandomChoice(size(p1));
    for i = pivot to size(p1) do
        child[i] = p2[i];
    end
    return child;
end
```

Algorithm 10: Two-point crossover

```
Function TwoPoint
    Input: p1, p2
    Output: child
    child ⟵ p1;
    pivot_1, pivot_2 = RandomChoice(size(p1), 2);
    for i = pivot_1 to pivot_2 do
        child[i] = p2[i];
    end
    return child;
end
```

Algorithm 11: Uniform crossover

```
Function Uniform
    Input: p1, p2
    Output: child
    child ⟵ EmptyVector(size(p1));
    for i = 0 to size(p1) do
        if p1[i] equals p2[i] or Random() ≤ 0.5 then
            child[i] = p1[i];
        end
        else
            child[i] = p2[i];
        end
    end
    return child;
end
```

Algorithm 12: Flip bits mutation

```
Function FlipBits
    Input: individual, swaps
    Output: individual
    positions ⟵ RandomChoice(size(individual), swaps);
    for p ∈ positions do
        individual[p] = individual[p] + 1  (mod 2);
    end
    return child;
end
```

For visual examples of the above operators, refer to section 2.2 of chapter 2.

The last procedure to address in this section is the population replacement that takes place at the end of every generation. With all the details we have previously taken care of, its implementation is quite straightforward:

Algorithm 13: Population replacement

```
Function Replace
    Input: population, size
    Output: next population
    next population ⟵ ∅;
    for i = 0 to size do
        next population ⟵ next population ∪ population[i];
    end
    return next population;
end
```

As we can see, since the population is already sorted by front and by crowding distance within the same front, the only thing we have to do is to apply a cutoff to keep the population size stable.

There is, nevertheless, a topic we have not covered with all the pseudocode of this chapter: the fitness or objective functions. In accordance with the design of the algorithm, they are based on minimization rather than maximization. They will measure the success of all our algorithms from now on, and they are the following:

- Simplicity: measures the number of active features. It is just the count of non-zero positions.
- Test set error: trains a model with the training set and evaluates the accuracy on the test set. Its range is between 0 and 1.
- Cross-validation error: splits the training set in n sections, using at each time one of them to test and the rest for model training. The final value is the arithmetic mean of all testing errors. Its range is between 0 and 1.
- Cohen's Kappa error [17]: similar to the test set error, but considering the probability of classifying correctly by chance. It is computed as:

$$\kappa = \frac{p_0 - p_c}{1 - p_c}$$

  Where p_0 is equivalent to the test set error, and p_c is the sum of the probabilities of random agreement for all possible classes.

Having now outlined the feature selection process, in the next chapter we will devise a similar path to undertake the optimization of a neural network.

6 NEURAL NETWORK OPTIMIZATION

The definition of hyperparameter is often fuzzy. In this work, we view hyperparameters as those parameters which are fixed before training a model, because there are no direct rules to infer them from the available data. In fact, we probably would not need them at all if we had enough data, since the samples would tell us everything that is to be known about a given problem. As we will almost never find ourselves in such a favorable situation, there exists a real need to find the right combination of hyperparameters for the circumstances at hand.

One could understand feature selection as a form of hyperparameter optimization, since it is in a way fixing a part of the model—the entry point of the data. Although it is undoubtedly a key tool when dealing with an absurd quantity of decision variables, whether it is hyperparameter optimization or not, it already has a section in its own right. From here on, we will focus on tuning aspects specific to neural networks, such as the overall structure, the learning rate, the number of epochs in training or the dropout rate [18].

Perhaps the most common technique for this task is the *grid search*. In grid search, we manually specify a set of values for each hyperparameter, and then the algorithm tries every possible combination. While efficient in low-dimensional spaces, it can also suffer from the curse of dimensionality. However, it is *embarrassingly parallel* in exchange, for the combinations are independent from one another.

When the above method becomes too resource-intensive, random search [19] comes into play. It was recently shown that a random search is capable of achieving similar results within a fraction of the time otherwise consumed by a grid equivalent; furthermore, it could yield even better outcomes if granted the same computation power.

Finally, if we look at quality-guided searches, we find classic algorithms such as *simulated annealing* [20], *particle swarm optimization* [21] or, again, genetic algorithms [22].

For scope constraints we will make use of genetic algorithms as our main optimization technique, carrying on our NSGA-II implementation from the feature selection phase. In general, it is computationally more expensive than random search—future work could tackle the problem from this point of view—, but in turn it has the genetic operators to leverage.

In the following two sections we will describe two separate optimizations: first, we will find an approximate structure (hidden layers and their sizes); then, we will tune the learning and dropout rates and

the number of training epochs. The reason for this is that their nature is different—the former deals with a variable number of parameters that define a whole, whereas the latter comprises exactly three parameters that are not conceptually linked together. Besides, joining these two parts would entail an overly convoluted search process.

Now, let us begin without further delay. We will reuse the NSGA-II framework (Algorithm 1), as well as Algorithms 3, 4, 5, 6, 7 and 13.

6.1 STRUCTURE OPTIMIZATION

We define the structure of a neural network as its configuration of layers and units per layer. To find out which combination is more appropriate, we will input a maximum number of layers—fixing the other parameters so that comparisons are fair—and the genetic algorithm will give us its rough estimation of what works best.

The only change with respect to Algorithm 1 is that it will receive the input size and the maximum number of hidden layers instead of the maximum number of features.

The population initialization will create a heterogeneous set of neural network structures which share one trait: they are widest in the middle layer and narrower towards the first and last layers. From this point on, the evolution has freedom to converge to other shapes.

Algorithm 14: Initialization of a population of structures

```
Function Initialize
    Input: population size, input size, max hidden
    Output: population
    population ⟵ ∅;
    sizes ⟵ RandomChoice(1, max hidden, population size);
    for i = 0 to population size do
        ind ⟵ EmptyVector(sizes[i]);
        ind[0] = RandomChoice(input size, 1.75*input size, 1);
        if sizes[i] > 1 then
            middle = sizes[i]/2 + 1;
            for j = 1 to middle do
                ind[j] = ind[j-1] + RandomChoice(ind[j-1], 1);
            end
            for j = middle to sizes[i] do
                ind[j] = ind[j-1] - RandomChoice(ind[j-1], 1);
            end
        end
        population ⟵ population ∪ {ind};
    end
    return population;
end
```

The initialization also has to consider the case where one or more layers in the second half reach a null or negative unit count. The correction will depend on the particular implementation.

Another procedure that undergoes slight modifications is the offspring creation. Given that structures are highly cohesive wholes, the crossover is not as useful and so the mutation takes a step forward in relevance: it is now independent from a successful crossover taking place, and contributes directly to the offspring pool (compare with Algorithm 8):

Algorithm 15: Offspring creation in structure optimization

```
Function CreateOffspring
    Input: parents
    Output: offspring
    offspring ⟵ ∅;
    for i = 0 to size(parents) do
        if Random() ≤ crossover probability then
            p1, p2 = RandomChoice(parents, 2);
            child ⟵ Crossover(p1,p2);
        end
        else if Random() ≤ mutation probability then
            p = RandomChoice(parents, 1);
            child ⟵ Mutation(p);
        end
        offspring ⟵ offspring ∪ {child};
    end
    return offspring;
end
```

Notice that the size of the offspring population is now determined by the sum of the two probabilities. The mutation will typically occur more often, thus making the algorithm resemble a random search with guiding criteria. Also, to improve the exploration we could have different types of mutation and alternate between them at random.

In Algorithms 16, 17 and 18 we can see the possibilities at our disposal at the time of writing.

The first one, *Midpoint crossover*, takes the ascending half of one parent and joins it with the descending half of the other parent.

The second one is the *Single Layer mutation*, which alters the number of units of a given layer and compensates the change by distributing the opposite operation among the other layers. It uses a magnitude modifier and a value from a normal distribution. This makes it a shape mutation rather than a size mutation.

The third one consists in scaling the whole network evenly by adding or subtracting a certain number of neurons at every layer—hence its name, *Uniform Scaling mutation*.

Algorithm 16: Midpoint crossover

```
Function MidPoint
    Input: p1, p2
    Output: child
    m = FindMiddleLayer(p1);
    n = FindMiddleLayer(p2);
    child ⟵ ∅;
    for i = 0 to m do
        child[i] = p1[i];
    end
    for i = 0 to size(p2) − n do
        child[m+i] = p2[n+i];
    end
    return child;
end
```

Algorithm 17: Single Layer mutation

```
Function SingleLayer
    Input: individual, magnitude
    Output: individual
    layer = RandomChoice(size(individual),1);
    change = individual[layer] * magnitude *
    RandomNormal();
    compensation = change/(size(individual)-1);
    individual[layer] = individual[layer] + change;
    for i = 0 to size(individual) do
        if i ≠ layer then
            individual[i] = individual[i] − compensation;
        end
    end
    return individual;
end
```

Algorithm 18: Uniform Scaling mutation

```
Function UniformScale
    Input: individual, magnitude
    Output: individual
    sign = 1 if Random() ≤ 0.5 else −1;
    change = Sum(individual) * magnitude / size(individual);
    for i = 0 to size(individual) do
        individual[i] = individual[i] + change * sign;
    end
    return individual;
end
```

6.2 TRAINING OPTIMIZATION

Now that we presumably have an idea of what sort of structure our neural network needs, it is time to take a further step to make the most of it. Since in structure optimization the remaining hyperparameters are chosen to speed up training times, it is unlikely that they already have the best values for full performance.

Let us take a closer look at the hyperparameters of this section:

- Learning rate: it dictates the fraction of the measured error that is used to correct the network (thus being between 0 and 1). A small one makes the training slower but more reliable, and a high one does the opposite. It is fundamental to find a value that allows a fast training without skipping over promising error minima.
- Number of epochs: it determines how many weight readjustments are performed before the training stops. Too few cause underfitting, while too many can produce overfitting.
- Dropout rate: it is a technique that disables a portion of the neurons (randomly selected) at each training epoch. As a result, it is a regularization method because the different sub-models that try to learn from the data have less predictive power. We need to tune it so that the final neural network is neither too powerful nor too simple for the task.

An individual in this algorithm will have a value for each of these three fields; for clarity, we will call them `ind.lr`, `ind.epochs` and `ind.dropout`, respectively.

Like we did in last section, we will use Algorithms, 3, 4, 5, 6, 7 and 13. Additionally, we want mutations to happen independently from crossovers, so we can use Algorithm 15 too. The NSGA-II framework (Algorithm 1) is modified to receive the maximum number of epochs and the ranges for the other two hyperparameters.

The initialization in this case is fairly trivial, as seen in Algorithm 19.

Algorithm 19: Initialization in training optimization

```
Function Initialize
    Input: population size, max epochs, lr range, dropout
           range
    Output: population
    population ⟵ ∅;
    for i = 0 to population size do
        ind.lr = Random(lr range);
        ind.dropout = Random(dropout range);
        ind.epochs = RandomChoice(1, max epochs, 1);
        population ⟵ population ∪ {ind};
    end
    return population;
end
```

Because the elements of each individual are independent numbers, the genetic operators are also straightforward. For example, one could perform a simple arithmetic mean:

Algorithm 20: Arithmetic crossover

```
Function Arithmetic
    Input: p1, p2
    Output: child
    child.lr = (p1.lr + p2.lr) / 2;
    child.dropout = (p1.dropout + p2.dropout) / 2;
    child.epochs = (p1.epochs + p2.epochs) / 2;
    return child;
end
```

If we view an individual as an array of values, we can also use n-point crossovers. Algorithm 21 describes a fixed Single-point crossover in which the child inherits the epochs from the first parent and the learning and dropout rates from the second parent. One could choose them in any order or make it randomized.

Mutations can make use of a normal distribution with our own center and standard deviation to introduce a variation in all values (Algorithm 22) or in just one of them. They can also target the dropout rate in isolation (Algorithm 23), given that it is more a technique rather than an intrinsic part of neural network training.

Algorithm 21: Single-point crossover in training optimization

Function *SinglePoint*
  Input: p_1, p_2
  Output: child
  child.lr $=$ p_2.lr;
  child.dropout $=$ p_2.dropout;
  child.epochs $=$ p_1.epochs;
  return child;
end

Algorithm 22: Gaussian mutation

Function *Gaussian*
  Input: ind, center, std
  Output: ind
  ind.lr $=$ ind.lr * RandomNormal(center, std);
  ind.dropout $=$ ind.dropout * RandomNormal(center, std);
  ind.epochs $=$ ind.epochs * RandomNormal(center, std);
  return ind;
end

Algorithm 23: Dropout mutation

Function *Dropout*
  Input: ind, magnitude
  Output: ind
  sign $= 1$ **if** Random() ≤ 0.5 **else** -1;
  change $=$ sign * magnitude;
  ind.dropout $=$ ind.dropout + change;
  return ind;
end

We have reviewed the key points of how the structure optimization (section 6.1) and training optimization (section 6.2) algorithms work. In the next part, we will apply these concepts and extract some conclusions.

Part III

EXPERIMENTS AND CONCLUSIONS

7

EXPERIMENTAL RESULTS

In this chapter, we start by detailing in section 7.1 the choice of technologies in order to obtain experimental results. After that, these results will be discussed in order of application in sections 7.2 (feature selection), 7.3 (structure optimization) and 7.4 (learning optimization). By the end, we will have empirical evidence to point us in promising directions, which we will subsequently address when we talk about conclusions and future work.

7.1 SOFTWARE AND HARDWARE

7.1.1 *Software*

The first decision to make is which programming language to use. `Python` is the choice for the following reasons:

- Previous experience with the language in web and machine learning applications.
- Popularity of the language, which is a good indicator of community support. According to the *StackOverflow* 2018 Survey [1], it is one of the most popular languages, and more so if we compare it with those commonly associated with machine learning in the last few years.
- Popularity of its machine learning and deep learning frameworks. Well-established frameworks include `Scikit-learn` [2], `Caffe` [3], `TensorFlow` [4] and `Theano` [5]. The last two of them also function as backends for the high-level neural networks API `Keras` [6]. If we look at the number of stars in their *GitHub* repositories, we can see that they are widely acknowledged by the community.

1 *StackOverflow* 2018 Survey: Most popular technologies
2 `Scikit-learn` GitHub repository
3 `Caffe` GitHub repository
4 `TensorFlow` GitHub repository
5 `Theano` GitHub repository
6 `Keras` GitHub repository

The next step is choosing the tools to support our work. Since one of the goals of this project was to learn about optimization techniques, the genetic algorithm has been implemented from scratch, although there are alternatives like `DEAP` [7] if one wishes to avoid the additional development effort.

Data operations become faster and easier with `Numpy` [8]. This will allow us to manage populations in genetic algorithms, as well as perform basic operations in a vectorized way whenever we need them. It is also fully compatible with the other libraries.

Building machine learning models from scratch too is understandably out of the question. For this reason, we will rely on `Scikit-learn` for general machine learning algorithms and metrics, and on `Keras`—with its default `TensorFlow` backend—for neural networks.

Lastly, many charts will be generated using `R` [9], which provides simple and powerful functionality for this task.

All the source code and experimental results can be found in its dedicated *GitHub* repository [23].

7.1.2 *Hardware*

We can make a distinction in this regard between the main development system, used for testing and debugging, and the dedicated servers for full-scale experimentation:

- Development system:
  - Intel® Core™ i5-3470 CPU @ 3.20GHz, 8GB DDR3.
  - NVIDIA GeForce® GTX 960, 2GB GDDR5.
- First dedicated server:
  - Two Intel® Xeon® E5-2620 v2 @ 2.10GHz, 32GB DDR3.
  - NVIDIA Tesla® K20c, 5GB GDDR5.
- Second dedicated server:
  - Intel® Xeon® E5-2620 v4 @ 2.10GHz, 32GB DDR4.
  - NVIDIA Tesla® K40m, 12GB GDDR5.
- Third dedicated server:
  - Two Intel® Xeon® E5-2620 v4 @ 2.10GHz, 32GB DDR4.
  - NVIDIA Tesla® K40m, 12GB GDDR5.

7 `DEAP` GitHub repository
8 `Numpy` GitHub repository
9 `R` Project website

7.2 FEATURE SELECTION

Determining the right configuration for the genetic algorithm—or rather, even one that is good enough—is no trivial task. Early experimentation seemed to point to a high crossover probability, but especially to a high mutation probability (ultimately set to 1). We will elaborate on that soon.

Let us start by comparing three different crossover operators: *Uniform*, *Single-point* and *Two-point*. We will work with a population of 300 individuals and 150 generations. We will set a 0.9 crossover probability after which a mutation will always ensue. A maximum of 50 active features will be allowed in each individual. The fitness criteria will take into account the Kappa value (for test accuracy) and a 5-fold cross-validation (for generalization assessment) measured for Logistic Regression.

We will do an initial test run on each subject (104, 107 and 110).

Figure 7.1 displays the evolution of the mean Kappa error for all crossover operators in all three subjects.

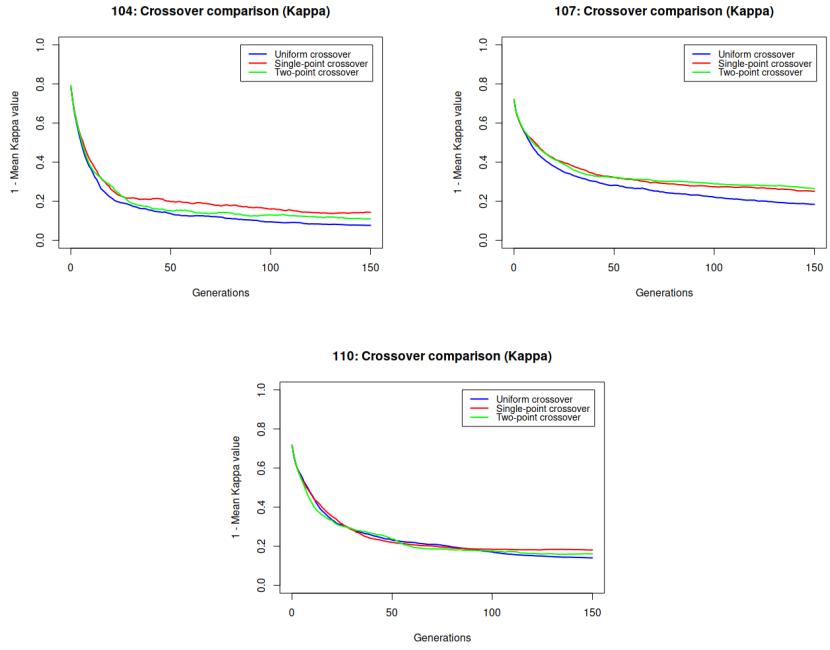

Figure 7.1: Comparison of Kappa loss evolution over time with different crossover operators.

We can identify appreciable differences between the three curves: the blue one (uniform crossover) shows consistently better results than the other two, and the red one (single-point crossover) appears to be the worst.

Let us move on now to the same type of chart but with the cross-validation error (Figure 7.2). Again, the uniform crossover operator achieves the top performance across all individuals and the single-point crossover operator often lags behind.

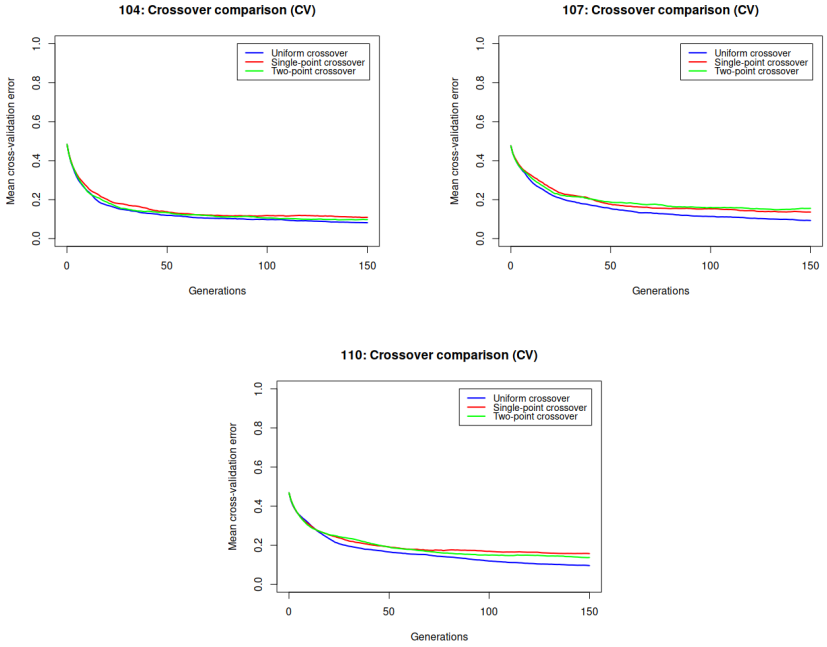

Figure 7.2: Comparison of cross-validation loss evolution over time with different crossover operators.

We seem to be spotting a trend that depends on the crossover operator. However, is it wise to extrapolate from only one test run? The answer is no: the running times allow us to repeat the experiment several times and find out whether their differences are statistically significant.

As a compromise between quantity of samples and time expended, we will analyze 15 samples per crossover operator using their Kappa error values. Table 7.1 shows the resulting values for all possible combinations:

Subject	Uniform	Single-point	Two-point
104	0.06534 ± 0.0074	0.08619 ± 0.0102	0.07941 ± 0.0119
107	0.15202 ± 0.0132	0.18004 ± 0.0137	0.17725 ± 0.0150
110	0.14829 ± 0.0138	0.17131 ± 0.0158	0.16569 ± 0.0187

Table 7.1: Comparison of average Kappa error values for the three subjects and the three crossover operators.

The average performance of each operator seems to be what we expected. Additionally, the uniform crossover appears to produce slightly more stable results, judging from the standard deviation. Next, and not making assumptions about normality, we will perform a *Kruskal-Wallis* test to see if their differences are worth considering. The *p-values* are displayed in Tables 7.2, 7.3 and 7.4, with values below 0.05 representing meaningful differences (95% confidence interval).

104	Single-point	Two-point
Uniform	$p = 0.000027$	$p = 0.000835$
Single-point		$p = 0.056282$

Table 7.2: Comparison of p-values for the crossover operators (subject 104).

107	Single-point	Two-point
Uniform	$p = 0.000023$	$p = 0.000104$
Single-point		$p = 0.678133$

Table 7.3: Comparison of p-values for the crossover operators (subject 107).

110	Single-point	Two-point
Uniform	$p = 0.000454$	$p = 0.006561$
Single-point		$p = 0.299489$

Table 7.4: Comparison of p-values for the crossover operators (subject 110).

As we can see, the single-point and two-point crossovers show no statistically significant differences in their recorded results.

The computational impact of either operator is negligible against the much more intensive fitness evaluations, so there is no reason not to consider the uniform crossover as our pick from now on. The last thing to do before moving on is attempt to explain why this happens.

On one hand, n-point crossovers merge large blocks from both parents; this leads to a not-so-optimal information transfer when trying to pick a handful of features from a pool of 3600.

On the other hand, the uniform crossover is often said to be very disruptive, due to randomly choosing elements for which the parents do not agree. However, when only a small portion k of features is active at a time, the disruption is at most $2k$ elements (the rest are inactive features); on top of that, features in which both parents agree are always kept. This makes the uniform crossover excel at *exploitation* while also helping in *exploration*. Together with frequent mutations, this is probably the key of the performance gain of the algorithm.

Another aspect to look into is the effect of population sizes and number of generations. Bigger populations should introduce greater exploration possibilities, while more generations should allow the genetic algorithm to converge to even better solutions. Nevertheless, an increase in these parameters has a direct influence in computation times, so we will have to check if the improvement is worth the effort.

We were working with a population size of 300 individuals and a number of generations equal to 150. The first alternative we propose is a tradeoff between a bigger population and less generations: 500 and 100. The second alternative is an increase in both: 800 individuals and 200 generations. Like we did before, we will start by observing what happens in a single run. In Figure 7.3 we can take a look at the behavior of the genetic algorithm in terms of Kappa loss.

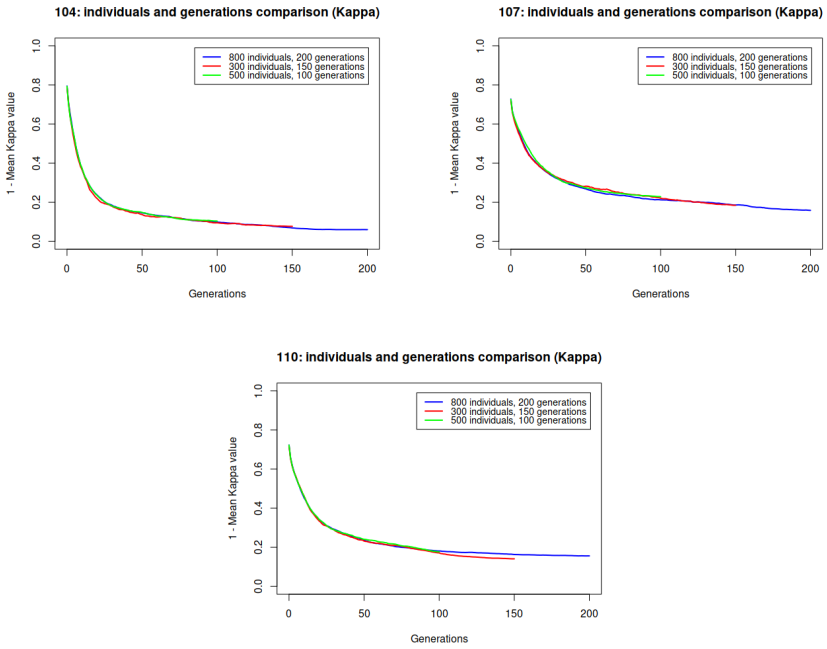

Figure 7.3: Comparison of Kappa loss evolution over time with different configurations of population and generations.

There is one definite conclusion we can draw: the number of generations is preventing the emergence of better individuals. The curves follow similar paths until they are progressively stopped by the generation limit. If we take a look at Figure 7.4, the same phenomenon is taking place for cross-validation loss, which does not surprise us.

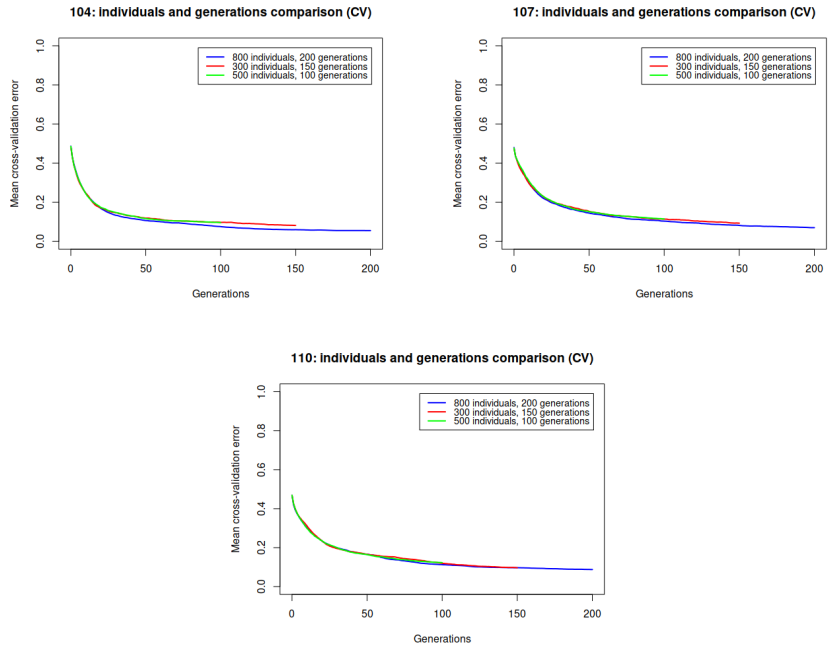

Figure 7.4: Comparison of cross-validation loss evolution over time with different configurations of population and generations.

It is clear that we need a relevant analysis like that of crossover operators. We will keep the number of individuals intact to grant more diversity, but we have to confirm the importance of extending the evolutionary process. For all this, another 15 algorithm runs for each alternative will be used to make statistical claims.

First, let us put the average performances alongside one another in Table 7.5:

Subject	300-150	500-100	800-200
104	0.06534 ± 0.0074	0.06310 ± 0.0077	0.05127 ± 0.0067
107	0.15202 ± 0.0132	0.16830 ± 0.0145	0.12122 ± 0.0126
110	0.14829 ± 0.0138	0.15448 ± 0.0069	0.13369 ± 0.0083

Table 7.5: Comparison of average Kappa error values for the three subjects and the three configurations.

We see that taking the population size up to 800 and the generations up to 200 yields finer average results. It is also noticeable that a bigger population appears to matter less when the number of generations is not accordingly increased.

Tables 7.6, 7.7 and 7.8 show the Kruskal-Wallis p-values for every pair of combinations. Again, we work with a 95% confidence interval, which means that values below 0.05 point to statistically significant differences.

104	300-150	500-100
800-200	$p = 0.000144$	$p = 0.000301$
300-150		$p = 0.884484$

Table 7.6: Comparison of p-values for the evolutionary configurations (subject 104).

107	300-150	500-100
800-200	$p = 0.000016$	$p = 0.000005$
300-150		$p = 0.002441$

Table 7.7: Comparison of p-values for the evolutionary configurations (subject 107).

110	300-150	500-100
800-200	$p = 0.003392$	$p = 0.000005$
300-150		$p = 0.034037$

Table 7.8: Comparison of p-values for the evolutionary configurations (subject 110).

The first thing we notice is that the combination 800-200 is consistently different from the other two. The differences between 300-150 and 500-100 are not so clear at times (see individual 104), but we could say that in a general case the approach with more generations can arrive at better solutions.

We find ourselves in a quandary between prioritizing computation times or results. The running times are certainly higher with 800-200, but they are still under an hour for a single subject, which means we can do a lot in a full day. On the side of quality, an improvement of just a few percentage points at this level can be invaluable. Given all this, we will keep the 800-200 configuration and make it our baseline for our forthcoming comparisons against neural networks.

Another topic to cover is the choice of fitness metrics. In particular, we can foresee that a substantial number of features will lag the already slow training in neural networks. One way to solve this could involve adding a third fitness function to favor individuals with fewer features. Let us proceed to briefly evaluate this possibility.

We will use a straightforward simplicity measure that returns the number of active features of an individual. The goal of the algorithm will be, like with the other two, to minimize it. Sample experiments with and without this simplicity measure can be observed in Figure 7.5.

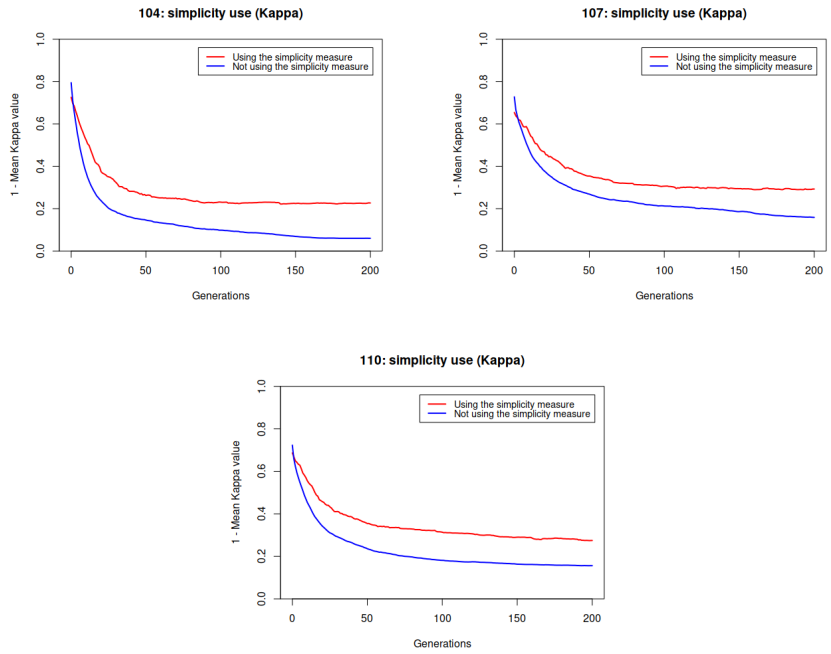

Figure 7.5: Comparison of Kappa loss evolution over time with and without the simplicity measure. A similar phenomenon occurs with the cross-validation loss.

The reported difference in performance is enormous. There is no need to do any statistical tests to realize that it will keep happening. The explanation is as easy as it gets: the simplicity measure hinders the progress of the other two criteria to the point of ultimately stalling it. In fact, for all we know there might be individuals with just one feature and abysmal accuracy in the first Pareto front.

In order not to twist the original logic of the algorithm, we will just use the feature cap as a means to keep the number of features under control—more research could be done about where the optimal cap lies. The other two fitness measures will still be used, since they provide valuable assessment about accuracy on unseen data and generalization capability.

To wrap up this section, the last improvement we will introduce is the choice of models for the fitness functions. One could argue that introducing a better model would only confirm that the model is more suitable for the problem and not that it can find better features; however, since it is not possible to formally prove or disprove it, we will give it a go.

This time, we will be measuring Logistic Regression against Support Vector Machine (SVM). Figure 7.6 shows the Kappa loss of a tentative run just to know if the comparison made any sense in the first place.

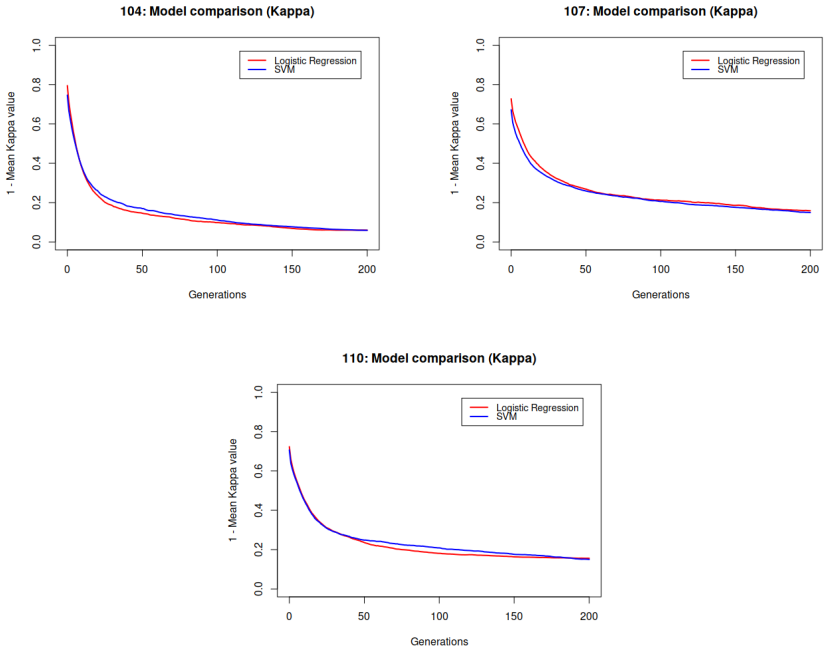

Figure 7.6: Comparison of Kappa loss evolution over time with Logistic Regression and SVM.

From what we see, it is not clear which model to select. SVM tends to converge more steadily, while the evolution of Logistic Regression tends to be steeper at the beginning. Neither one of them finish with clear advantage respect to the other, though.

Perhaps the evolution of the cross-validation loss can help us assess their distinctive qualities. Figure 7.7 contains the analogous plot. In it, we can observe how towards the right end of the curve the SVM-driven experiment surpasses the other one in average performance. This is a sign that SVMs could have an edge on Logistic Regressions for this dataset.

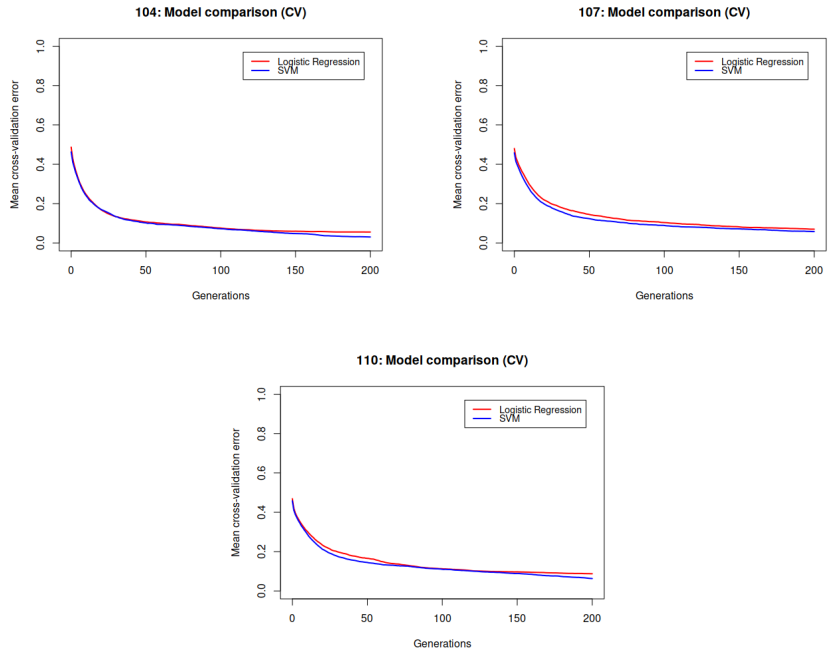

Figure 7.7: Comparison of cross-validation loss evolution over time with Logistic Regression and SVM.

Like we have already done twice, it is natural to record several experiments to confirm or dismiss what we have just seen. Making use of the best algorithm configurations found until now, Logistic Regression and SVM (with $C = 0.1$) will be tested against each other by averaging their best Kappa loss from 15 runs. Table 7.9 holds the numbers.

Subject	SVM	LogReg
104	0.05127 ± 0.0103	0.05128 ± 0.0067
107	0.09546 ± 0.0093	0.12122 ± 0.0126
110	0.11796 ± 0.0095	0.13369 ± 0.0083

Table 7.9: Comparison of average Kappa error values for the three subjects and the two models in question.

It is quite apparent that SVMs provide yet another push in classification quality—it is also expected that cross-validation measures follow the same pattern, given that they triggered this comparison.

In the next page we can find Table 7.10, which contains the p-values for the Kruskal-Wallis test for statistically significant differences.

Logistic Regression against SVM		
104	**107**	**110**
$p = 0.917098$	$p = 0.000025$	$p = 0.000060$

Table 7.10: Comparison of p-values for Logistic Regression and SVM in all test subjects.

Again, the test shows that there is often a relevant accuracy boost when using SVM instead of Logistic Regression (in subject 104 we might have reached the full potential). Although we do not show them here, the cross-validation differences are noticeable too.

It is also true that this change has taken a toll on the running time of the algorithm. Nevertheless—and following the same logic as when we increased population sizes and number of generations—, the improvements in quality of the solutions are crucial at this level. Since we can still run a couple dozen experiments a day (counting the three test subjects), we can safely sacrifice efficiency in favor of further gains.

We have finally completed our review of feature selection enhancements. Although by now it should go without saying that feature selection is essential in our problem, Table 7.11 puts our best results so far (in terms of Kappa loss) against classifiers using all 3600 features.

	Feature selection		No feature selection	
Subject	**Log Reg**	**SVM**	**Log Reg**	**SVM**
104	0.042246	0.033781	0.279954	0.305215
107	0.101059	0.075823	0.327885	0.328009
110	0.117981	0.101107	0.404468	0.404545

Table 7.11: Best Kappa loss scores of feature selection versus no feature selection.

Notice that we list both Logistic Regression and SVM to introduce a bit more information: even though the former behaves better with no feature selection, it ultimately fails to equal the latter after both are optimized under the same conditions.

These results constitute the baseline for the next two sections, which will be dedicated to the optimization of neural networks using the best features we have been able to obtain.

Since the accuracy is already unexpectedly good, our efforts will be focused on discerning whether there is still room for improvement with the use of more advanced models (such as neural networks) or the remaining gap is just an intrinsic limitation of the data. We will also discuss the pros and cons of applying neural networks to this specific dataset.

7.3 STRUCTURE OPTIMIZATION

This section covers the first step of the proposed neural network optimization workflow: the search for a good structure.

Before we start, it should be noted that, as opposed to feature selection—where time is not much of a concern—, from now on the experiments will be severely limited by the training process of neural networks. The decrease in population sizes and number of generations will be drastic (15 to 20 times less), but the running times will still be one order of magnitude higher if we cannot find any solutions. This is one of the reasons why we have put so much effort into finding the right features.

Having said that, let us begin by explaining the motivation for a two-step optimization. When we try to optimize several parameters at the same time, the combined search space is as big as the product of the sizes of the individual search spaces; this is often quite a lot, and the genetic algorithm has to work with accordingly bigger populations and evolution periods to make up for it. In an attempt to take away a part of this complexity, we can concede a (hopefully) small decrease in potential quality and try to optimize one (or a few) of those parameters in isolation.

If we agree to the aforementioned logic, the first point in question is how to split the process in a way that the disruption to the overall optimization is minimized. Let us take a look at what parameters—hyperparameters— we will be dealing with:

- Learning rate.
- Number of training epochs.
- Dropout rate.
- Structure (number of layers and units per layer).

It is possible to identify a pattern that divides them into two groups: the first three are discrete numbers that are adjusted once we have defined a model structure; the fourth is precisely that structure. Therefore, intuitively we may find a structure first and then use it to work out the best combination of the remaining hyperparameters.

Assuming what we have just said, we need to determine how to optimize the structure while also looking for ways to speed up the process. For this, we will bring up the experimental results of last section. The high accuracies of linear models suggest that our neural networks should not be very complex, because if we couple that with our small training sets they will most likely suffer from overfitting.

Now, let us see how we can turn this in our favor. Suppose a population of 50 individuals and 20 generations, with the objective functions being again the Kappa loss and the 5-fold cross-validation. The amount of neural network trainings for each function is given by:

- Kappa: $50 \times 20 = 1000$ trainings.
- 5-fold cross-validation: $5 \times 50 \times 20 = 5000$ trainings.

Which, summed up, yield a total of 6000 trainings.

Let us discuss the importance of each fitness measure in the process. The Kappa value uses the test set to give an estimate of how well the model will behave with unseen data. The cross-validation accuracy, however, assesses the generalization capability of the model (without using new data), which is directly tied to a low overfitting. Then, if we swap in a simplicity measure for the cross-validation function, we are still pushing in the same direction to an extent.

The proposed simplicity measure is the sum of all the neurons of the given structure (note that the time it takes to compute it is negligible). This way, we can factor out 5000 of the 6000 evaluations (83%); in a more general statement, we are leaving the load of the fitness evaluation at something close to $T_{op} = T_0 - (\frac{k}{k+1})T_0 = (\frac{1}{k+1})T_0$, where k is the number of cross-validation folds.

Again, we have to keep in mind that this is only a fast alternative to improve the efficiency of this first optimization step—which is not the most important one. Also, it is derived from our previous results in this specific problem and, while there are experiments telling us that the speedup is worth the absence of cross-validation, further comparative testing would be required. Such testing is unfeasible in this work due to the large amounts of time needed for cross-validation.

However, despite not being able to statistically analyze their differences, a single visual comparison contains a lot of information. Figure 7.8 shows the evolution of a sample Kappa-CV run against a sample Kappa-Simplicity run; the first one had 30 individuals and 20 generations, while the second one had 50 individuals (thus taking advantage of the speedup to have a bigger population).

The first thing we notice is that the algorithm does not produce as significant a quality boost as in feature selection. Instead, it starts at an already decent point and tries to make improvements by evaluating new structures. Because it is only one of several hyperparameters to tune, it is not unusual to observe little average performance gains in both alternatives. Still, if we take a look at peak accuracy, Table 7.12 shows that these preliminary results are promising.

Another aspect we can analyze is the time they take to reach their best average. Without making too many assumptions, it seems that the simplicity measure makes that time shorter, while with cross-validation we still find improvements later on. This suggests that we could add more individuals in exchange for less generations.

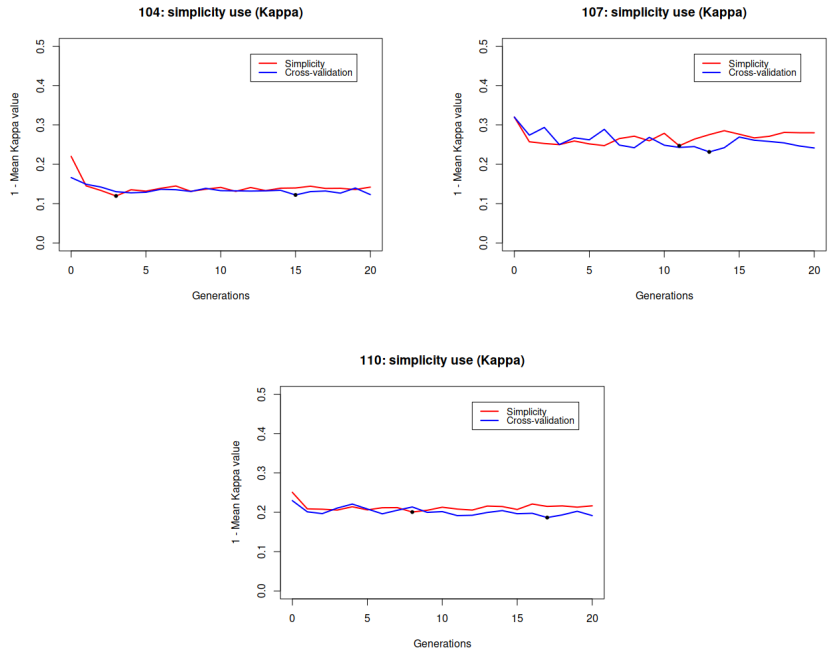

Figure 7.8: Comparison of mean Kappa loss evolution for both metrics. Notice that the vertical axis now has the range $[0, 0.5]$, as opposed to the previous $[0, 1]$ used in feature selection.

Subject	Cross-validation	Simplicity
104	0.09289	0.08456
107	0.16840	0.19395
110	0.15169	0.17690

Table 7.12: Comparison of top Kappa error values for both metrics.

As we can see in the table, the run employing simplicity trails behind a bit in peak performance but it manages to snatch the first place for subject 104. Knowing that it took much less time (1000 evaluations versus 3600), we can say that the tradeoff is more than fair.

After this intuitive exploration, further comparative testing would be required. Such testing is not feasible because cross-validation is computationally too heavy. However, at 30-40 minutes per subject, we can afford to average a number of simplicity-driven experiments.

Let us define the parameters first: we will use 60 individuals and 15 generations; the hidden layers will be limited to 4, which will produce a first population of rather complex models to be reduced by simplicity; the crossover (midpoint) and mutation (single layer and uniform scaling) probabilities will be 0.1 and 0.9; the neural network fitting epochs will be fixed to 25; finally, we will repeat the combination Kappa-Simplicity.

Table 7.13 shows the mean and peak Kappa loss for the three test subjects:

	Average	Best
104	0.09037 ± 0.0089	0.07598
107	0.21734 ± 0.0328	0.14319
110	0.20895 ± 0.0147	0.18543

Table 7.13: Average and best Kappa loss scores of ten simplicity-driven test runs.

Figure 7.9 supports the hypothesis that the pair Kappa-Simplicity is able to improve from an initial, unoptimized population. It is manifest that the average Kappa loss again converges soon, but we cannot risk losing an eventual peak in performance by diminishing the generation count excessively—early average convergence does not prevent top individuals from appearing.

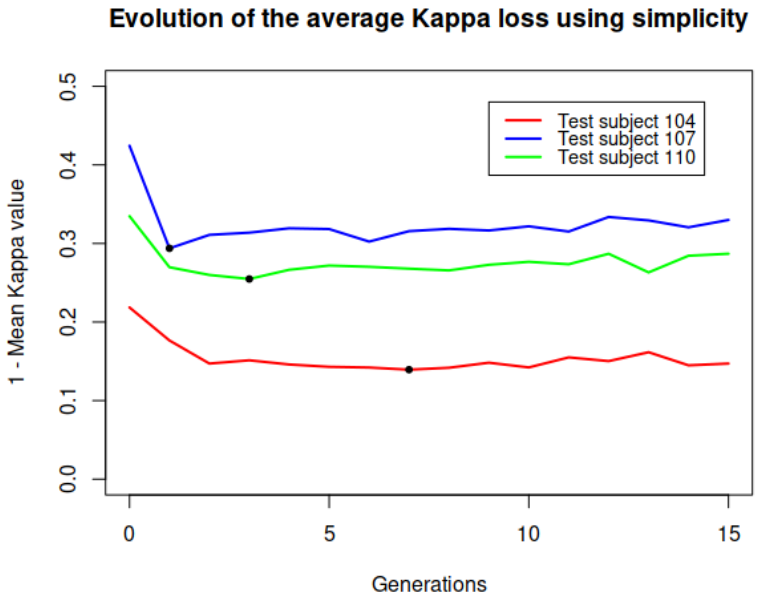

Figure 7.9: Evolution of the three experiments that produced the best peak accuracies.

In this section we have applied our previous knowledge to obtain neural network structures in short timespans. With them as a foundation, *learning optimization*—as we will call the last step—will be crucial in deciding if the whole process is able to produce competitive enough models.

7.4 LEARNING OPTIMIZATION

This arduous search process finds its conclusion in the optimization of the hyperparameters we left aside in last section: learning rate, dropout rate and number of training epochs. Let us begin by reviewing them:

- Learning rate: when samples are processed through a neural network during training, we can trace what made the network's output differ from the true label; additionally, the error can be quantified from the weights of the connections. The learning rate is the fraction of the error used to correct those weights. Therefore, its values are in the range $(0, 1]$.
- Dropout rate: dropout is a technique introduced recently [18] that consists in disabling a certain number of randomly chosen units during each training step. This serves as a regularization method, because the final model is like an ensemble of weaker neural networks that represent the sets of units that remained active at each step. It can range between 0 and 1, but the authors recommend trying from 0.2 to 0.5.
- Number of epochs: an epoch is an iteration over the entire dataset, meaning that in one epoch there can be multiple weight updates depending on how many samples are used to calculate the error. It ranges from 1 to as many as needed.

Notice that this time we do not have a clear way to avoid using cross-validation. Furthermore, since in this section we will seek to get the best possible performance out of our neural networks, we cannot afford to use shortcuts in place of direct quality measures. This will probably produce the following effect: the algorithm will show a smoother—almost monotonically decreasing—progression from its start point at the cost of a significant loss in speed.

At the time of the writing, we only had time for four experiments with each test subject. In Figure 7.10 we display the evolution of the three that returned the best results to show that the progression is indeed more stable. Table 7.14 contains the average and peak performances; it is plausible (and expected) that the peak results improve as we rerun the algorithm from more starting points.

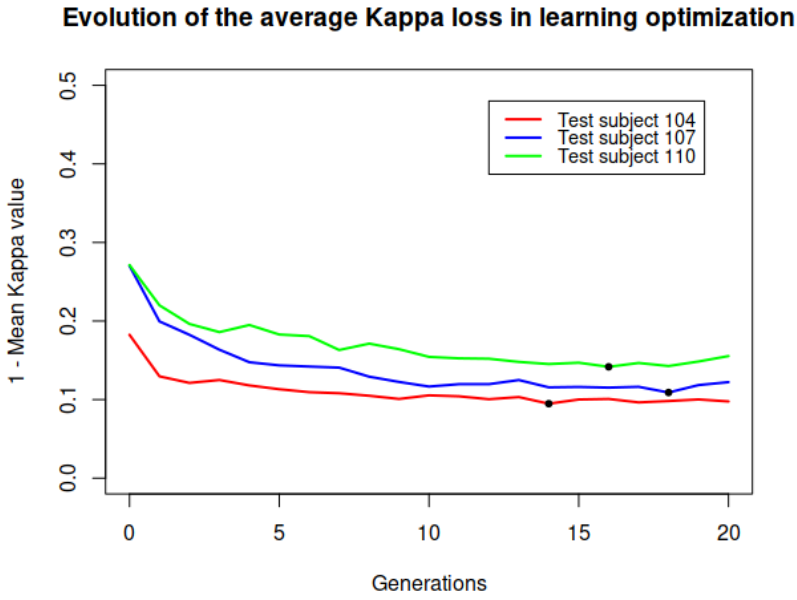

Figure 7.10: Evolution of the learning optimization algorithm runs that produced the best peak values.

	Average	Best
104	0.06756 ± 0.0119	0.05907
107	0.08636 ± 0.0106	0.07582
110	0.11794 ± 0.0069	0.10953

Table 7.14: Average and best Kappa loss scores of two learning optimization test runs.

As we can see in the table, the results we get are reasonably close to those obtained by SVM in the feature selection phase—and sometimes even better than Logistic Regression. Further experimentation should produce better peak results, although the extent of these gains is something yet to be tested.

Finally, we list the configurations that yielded these best results:

- Subject 104: 50 features, 1 hidden layer (110 units), 283 epochs, 0.027 learning rate, 0.0 dropout, ELU activation function.
- Subject 107: 46 features, 2 hidden layers (89 units, 102 units), 239 epochs, 0.048 learning rate, 0.05 dropout, ELU activation function.
- Subject 110: 50 features, 1 hidden layer (34 units), 286 epochs, 0.066 learning rate, 0.05 dropout, ELU activation function.

7.5 EFFICIENCY STUDY

Despite the fact that we have achieved excellent results throughout this work in terms of accuracy, one last aspect we should not disregard is efficiency. In the next few pages we will provide some insight into the relative time performance of multiple parts of the optimization process. We will start discussing speed in feature selection and afterwards we will move on to the much costlier neural network optimization and its more impactful potential efficiency improvements.

7.5.1 *Feature selection: Logistic Regression and SVMs*

The first comparison we are going to draw is the use of Logistic Regression against the use of SVMs at the feature selection stage. While the latter attains better average and peak performance, it is unknown if it happens because the model performs better for this problem or because it actually finds more suitable features. Leaving that topic for future research in this direction, we put side by side the averages of 5 runs with 300 individuals and 150 generations:

LogReg (s)	SVM (s)
1043.635	2220.472

Table 7.15: Comparison of average running times for Logistic Regression and SVM. Run in the second dedicated server.

We can confirm that SVMs significantly slow down the algorithm. However, without more information we cannot advise against them, because we might be losing better feature sets—and we have seen that they are the most important component of the process. In an attempt to alleviate this, we will proceed to find out if parallelism helps.

7.5.2 *Feature selection: sequential and parallel*

In order to make feature selection with SVM more scalable, parallelism would be very helpful. In this section, we will take a look at the speed gains when we increase the number of threads. For a start, we will see the average time the SVM-driven algorithm takes with the same 300 individuals and 150 generations if we use all threads (16):

Sequential (s)	Parallel (s)
2220.472	922.550

Table 7.16: Comparison of average running times for SVM with and without parallelism. Run in the second dedicated server.

The improvement is substantial, even surpassing the time achieved by sequential Logistic Regression. With that as a starting point, we will now plot the change in running times starting from one thread to all the second dedicated server is capable of. However, to make this experiment shorter, we will use 100 individuals and 50 generations instead. Figure 7.11 shows the evolution of running times.

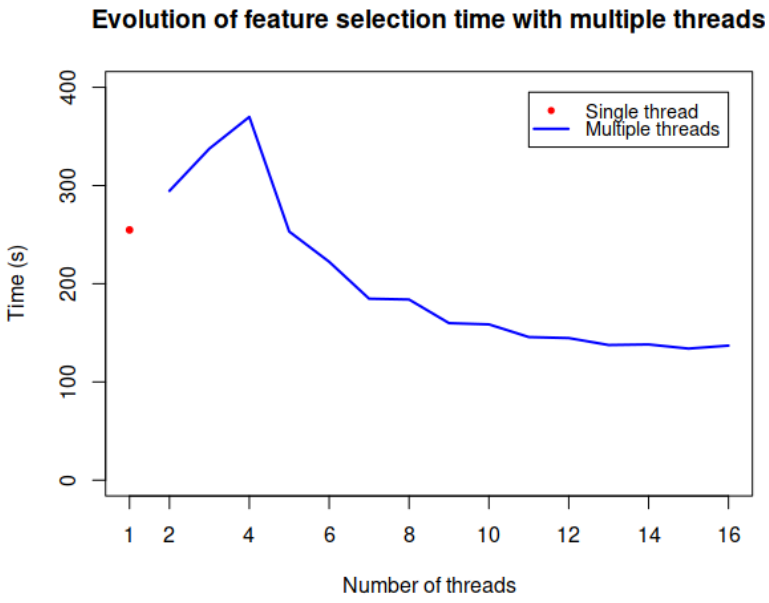

Figure 7.11: Evolution of CPU time for increasingly higher thread usage. Run in the second dedicated server.

Whereas using little parallelism worsens performance, we can observe how running times decrease steadily with the addition of more threads until it stabilizes at around half of the original. Heavier experiments are expected to take advantage of even higher thread counts.

7.5.3 *Neural network training: CPU and GPU*

As we saw in the experimentation section, cross-validation makes the whole process computationally very intensive. The first improvement that comes to mind is to use the GPU instead of the CPU to train neural networks. Since ours are rather simple, in Figure 7.12 we try to find the threshold from which the parallelism within a GPU starts to outperform the raw power of a CPU core.

The intersection seems to happen at around 4400 units in a single layer. Consequently, we would have to train models with the equivalent complexity of a 4400-unit single-layer network to be able to seize GPU parallelism—one could link this to multilayered networks by calculating the number of tunable weights, which hold the actual complexity of training the network.

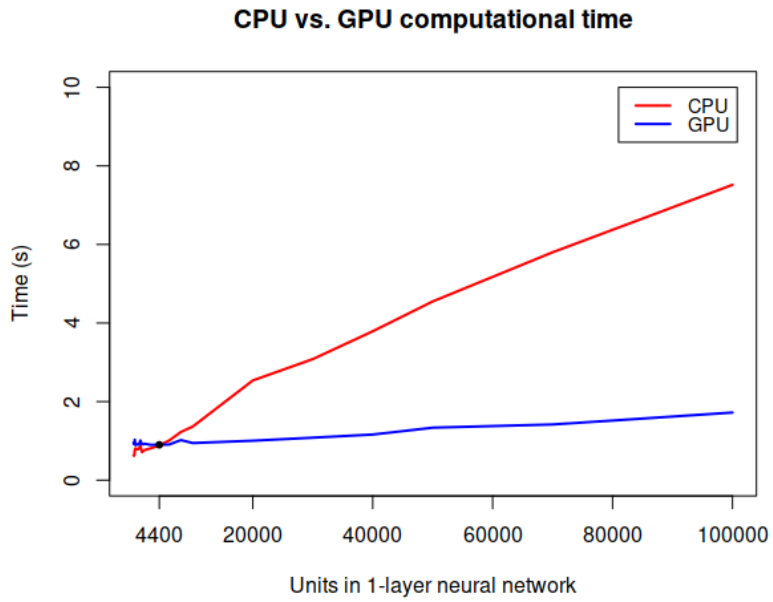

Figure 7.12: Evolution of CPU time and GPU time for increasingly bigger single-layer networks. Run in the first dedicated server.

Since in this particular problem we do not work with big enough networks, we have to find the speedup in multi-CPU parallelism, which we talk about in the last part of this efficiency study.

7.5.4 *Neural network training: sequential and parallel*

When using a GPU does not bring a notable performance enhancement, we turn to multi-CPU parallelism. In situations when we have several multithreaded cores, distributing training processes among them can be very beneficial.

We said that using cross-validation made the optimization algorithm severely slow in the case of neural networks. Let us see if multiple CPUs can make a difference. If we take a 10-individual, 5-generation setup and observe in Figure 7.13 what happens as we increase the thread count, we see that with just two threads the difference is massive. From that point on, subsequent gains are smaller but still noticeable until around 12 threads (the number of real CPU cores). Again, more ambitious experiments should make better use of it.

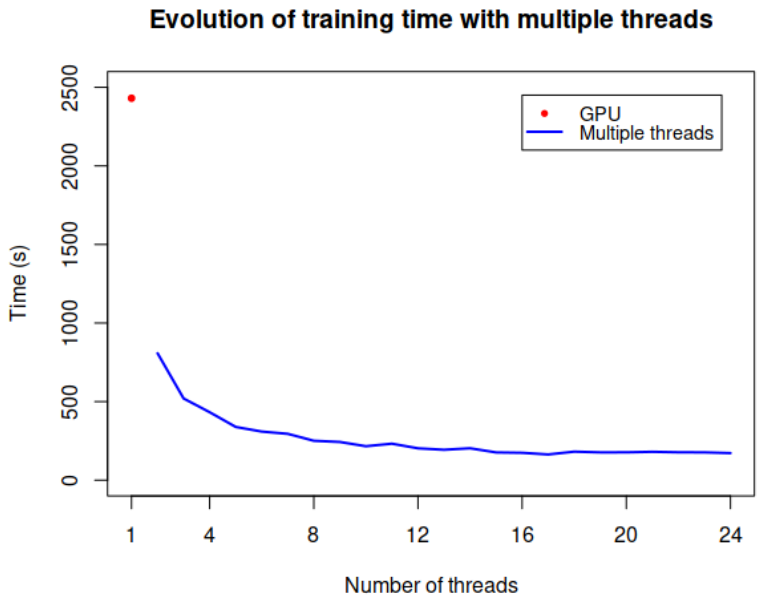

Figure 7.13: Evolution of multi-CPU time for increasingly higher thread counts. Run in the first dedicated server.

Unfortunately, technical issues have prevented full use of this last type of parallelism. The correction of these issues is left as a future task.

With this efficiency analysis we have reached the end of the chapter dedicated to experimental results. We have gone over the different steps and reasonings within them to reach significant quality in the final results. We have also discussed several ways in which the process might be made faster. In the next and closing chapter we will draw conclusions from the whole work and point to some promising research questions that have not found an answer here.

8

CONCLUSIONS AND FUTURE WORK

8.1 CONCLUSIONS

8.1.1 *Software developed*

At the end of this work we have a fully functional codebase [23] that is theoretically able to address any machine learning problem of the same nature as the one we have attempted to solve here; this means that we can perform a solid feature selection making use of well-established genetic operators and then pass the resulting features on to a two-step neural network optimization method of similar structure if the problem demands so.

The software is also able to leverage the implicit parallelism in GPUs via `TensorFlow` and the explicit process distribution to multiple CPU cores and threads, which—only lacking minor technical tweaks—makes for a scalable workflow that can be reused for larger datasets.

Additionally, the code is written in a modern language like `Python` and uses popular and up-to-date machine learning and deep learning libraries, which facilitates code maintenance, extensibility and longevity due to the wide community support.

We can safely state that we have accomplished the purpose of this work, since through the different optimization steps we have been able to choose in a non-arbitrary manner several hyperparameters found in neural networks: the structure (number of inputs and composition of the hidden layers) and three learning parameters (namely, the learning and dropout rates and the number of training epochs). Moreover, the process of designing and writing the necessary code has led to a superior understanding of how the involved technologies work, which was the remaining goal.

8.1.2 *The problem tackled in this work*

The dataset pertains to the area of BCI, and in particular it is aimed at telling apart the imagined movements of the left hand, the right hand and the feet. Decisive advances in this classification task have the potential to be useful in medical applications.

If we assume the accuracy obtained in this work as correct, these results unfold optimistic prospects in terms of practical applications. Although more detailed research is still needed, we can at least say this much. What is more, we have witnessed how SVMs—which are

not very intricate both in concept and in training—are capable of fulfilling the classification task to an outstanding level.

Using neural networks in this dataset is perhaps not the most efficient option. Nonetheless, it has allowed us to try the neural network optimization method and conclude that it is powerful enough to be useful.

8.2 FUTURE WORK

During the progress of this project many opportunities for further work have appeared. For clarity, we list the most important ones below:

- Find the right feature limit in feature selection. Currently, the algorithm tends to use almost as many features as it is allowed; this raises the question of where to put the cap in order to maximize accuracy without losing generalization to unseen data (including the test set).
- Perform detailed testing of the use of cross-validation against the use of simplicity in the structure optimization phase.
- Tune the SVM hyperparameter C in order to improve even more its current top accuracy. For this matter we could develop a modified version of the neural network learning optimization or we could use other techniques such as grid search or random search.
- Try more ambitious learning optimization setups (that is, more individuals and more generations) in order to find out if it yields better models.
- Find out if there is the possibity of feature transfer between different subjects. This would allow us to skip a big part of the process and make a possible real-world application easier to carry out.

Less relevant open issues include a better control over randomness in the code or making CPU parallelism a little more robust to increases in the scale of the experiments.

BIBLIOGRAPHY

[1] Jonathan R. Wolpaw et al. "Brain–Computer Interface Technology: A Review of the First International Meeting." In: *IEEE Transactions on Rehabilitation Engineering* 8.2 (2000).

[2] Christoph Guger, Shahab Daban, Eric Sellers, Clemens Holzner, Gunther Krausz, Roberta Carabalona, Furio Gramatica, and Guenter Edlinger. "How many people are able to control a P300-based brain–computer interface (BCI)?" In: *Neuroscience letters* 462.1 (2009), pp. 94–98.

[3] Kevin Gurney. *An Introduction to Neural Networks*. UCL Press Limited, 1997.

[4] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." In: *Nature* 521.7553 (2015), pp. 436–444.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." In: *International Conference on Computer Vision*. 2015.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition." In: *Computer Vision and Pattern Recognition*. 2016.

[7] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)." In: *arXiv* 1511.07289 (2015).

[8] Alberto Prieto, Beatriz Prieto, Eva Martinez, Eduardo Ros, Francisco Pelayo, Julio Ortega, and Ignacio Rojas. "Neural networks: An overview of early research, current frameworks and new challenges." In: *Neurocomputing* 214 (2016), pp. 242–268.

[9] Thomas Bäck. "Selective pressure in evolutionary algorithms: A characterization of selection mechanisms." In: *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence.* 1994.

[10] Edoardo Amaldi and Viggo Kann. "On the approximability of minimizing nonzero variables or unsatisfied relations in linear systems." In: *Theoretical Computer Science* 209.1-2 (1998), pp. 237–260.

[11] Isabelle Guyon and André Elisseeff. "An introduction to variable and feature selection." In: *Journal of machine learning research* 3.Mar (2003), pp. 1157–1182.

[12] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. "A fast and elitist multiobjective genetic algorithm: NSGA-II." In: *IEEE transactions on evolutionary computation* 6.2 (2002), pp. 182–197.

[13] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. "SPEA2: Improving the Strength Pareto Evolutionary Algorithm." In: *TIK-report* 103 (2001).

[14] Joshua Knowles and David Corne. "The pareto archived evolution strategy: A new baseline algorithm for pareto multiobjective optimisation." In: *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on Evolutionary Computation*. Vol. 1. IEEE. 1999, pp. 98–105.

[15] Ingrid Daubechies. *Ten lectures on wavelets*. Vol. 61. Siam, 1992.

[16] J Asensio-Cubero, JQ Gan, and Ramaswamy Palaniappan. "Multiresolution analysis over simple graphs for brain computer interfaces." In: *Journal of neural engineering* 10.4 (2013), p. 046014.

[17] Jacob Cohen. "A coefficient of agreement for nominal scales." In: *Educational and psychological measurement* 20.1 (1960), pp. 37–46.

[18] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: A simple way to prevent neural networks from overfitting." In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[19] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization." In: *Journal of Machine Learning Research* 13.Feb (2012), pp. 281–305.

[20] Ping-Feng Pai and Wei-Chiang Hong. "Support vector machines with simulated annealing algorithms in electricity load forecasting." In: *Energy Conversion and Management* 46.17 (2005), pp. 2669–2688.

[21] Shih-Wei Lin, Kuo-Ching Ying, Shih-Chieh Chen, and Zne-Jung Lee. "Particle swarm optimization for parameter determination and feature selection of support vector machines." In: *Expert systems with applications* 35.4 (2008), pp. 1817–1824.

[22] Frank Hung-Fat Leung, Hak-Keung Lam, Sai-Ho Ling, and Peter Kwong-Shun Tam. "Tuning of the structure and parameters of a neural network using an improved genetic algorithm." In: *IEEE Transactions on Neural networks* 14.1 (2003), pp. 79–88.

[23] *Hyperparameter optimization in Deep Neural Networks*. https://github.com/jleon95/UGR_TFG.

DECLARATION

JAVIER LEÓN PALOMARES con DNI 00000000X, actuando en su propio nombre y derecho,

DECLARA, BAJO SU RESPONSABILIDAD:

Que el Trabajo Fin de Grado presentado en la Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada, con fecha 18 de junio de 2018, titulado «*Hyperparameter optimization in Deep Neural Networks, In the context of EEG classification*» es original, no es copia ni adaptación de ningún otro trabajo, inédito, y no ha sido difundido por ningún medio, ni presentado anteriormente por quien suscribe o por otra persona.

Y para que así conste a los efectos oportunos, firmo la presente en Granada, a 18 de junio de 2018.

Javier León Palomares

Asimismo, yo, Javier León Palomares, alumno del Grado en Ingeniería informática de la Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada, con DNI 00000000X, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Javier León Palomares

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Thank you very much for your feedback and contribution.

Final Version as of June 18, 2018 (`classicthesis` version 4.4).