UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Degree as Computer Science Engineering and Mathematics

DEGREE WORK

Interpretable models in machine learning

Author: Fernando Villar Gómez
Advisor: Alberto Suárez González

Wednesday 10th June, 2020

All rights reserved.
No reproduction in any form of this book, in whole or in part
(except for brief quotation in critical articles or reviews),
may be made without written authorization from the publisher.

© June 2020 by UNIVERSIDAD AUTÓNOMA DE MADRID
Francisco Tomás y Valiente, n° 1
Madrid, 28049
Spain

Fernando Villar Gómez
Interpretable models in machine learning

Fernando Villar Gómez

PRINTED IN SPAIN

Prefacio

El origen de este trabajo se remonta a una conversación entre Alberto Suárez y yo muchos meses antes de que ni siquiera tuviéramos que preocuparnos por la asignación del Trabajo de Fin de Grado. Ya por aquel entonces el tema de las limitaciones de las redes neuronales demostró ser de gran interés para ambos, por lo que por mi parte no hubo dudas cuando Alberto me lo propuso.

Después de un tiempo, por azares del destino, el tema del TFG cambió, pero sólo por unos meses: al final, los dos volvimos a añorar aquel posible nuevo modelo que combinara árboles de decisión con redes neuronales, y terminamos por hacerlo realidad.

Aunque aún queda mucho trabajo pendiente para el futuro, la motivación de encontrar una alternativa real a las redes neuronales, tan predominantes en el presente, es lo que nos ha llevado a desarrollar el contenido de este trabajo, que esperamos que sea la primera piedra para provocar un cambio de paradigma en el que saber lo que se está haciendo empiece a recobrar la importancia que un día perdió en contra de, simplemente, hacerlo.

Fernando Villar Gómez

Agradecimientos

En primer lugar, me gustaría agradecer a mi tutor, Alberto Suárez, el empeño y el cariño que ha puesto en este trabajo desde el principio, permitiéndome desarrollarlo como más cómodo me he sentido y aportándome mucho conocimiento que me será de gran utilidad para el futuro. Y, cómo no, también por permitirme cambiar el tema del trabajo y así evitar desfallecer entre terribles sufrimientos.

También querría agradecer a mi grupo de compañeros de clase por hacerme la vida universitaria menos difícil y aburrida, además de ayudarme siempre que lo he necesitado: Santiago, Yujian, David, Antonio, Aitor, Adrián y Guillermo (incluso las discusiones merecieron la pena).

Tampoco quiero olvidarme de mis compañeros de QRR, que me han alegrado los días cuando el estrés podía conmigo, especialmente mis guías desde el primer día, Raúl y Jaime.

Por último, quiero recordar en estas líneas a los profesores que me han ayudado a finalizar esta experiencia de una forma un poco más satisfactoria: Tania, Yago, Andrei, Simone, Alberto, Dragan, Ana, el dúo Fernández, Bartolomé y muchos otros. Sin vosotros no sé si no hubiera sido posible, pero desde luego habría sido un camino mucho más tortuoso. Os lo agradezco de corazón.

Resumen

Desde hace unos años, la investigación y el desarrollo del aprendizaje automático se ha disparado con la revolución de unas redes neuronales que, tras la mejora de la potencia del hardware y la disponibilidad de grandes volúmenes de datos, se han convertido en la mayor esperanza en el campo de la inteligencia artificial. Sin embargo, parece que más recientemente los avances son menores, probablemente porque están llegando a su límite de precisión. Además, las redes neuronales presentan el inconveniente de ser una caja negra, dentro de la cual no es posible entender el modo en el que la red predice. Por tanto, a pesar de su precisión, no permiten sustentar mediante argumentos sus resultados. Los árboles de decisión, por otra parte, son un modelo muy interpretable, pero generalmente poco preciso. El objetivo de estudio de este proyecto es la creación de un modelo mixto entre las redes neuronales y el árbol de decisión: el árbol de decisión de hoja caduca. En este modelo de predicción se combina la estructura del árbol de decisión con la capacidad predictiva de la red neuronal, buscando un punto intermedio personalizable en la dicotomía entre precisión e interpretabilidad. Los resultados obtenidos tras el análisis de varios experimentos usando el árbol de decisión de hoja caduca muestran que, si bien se produce una pequeña pérdida en la precisión, esta pérdida no es lo suficientemente grande como para compensar la ganancia en interpretabilidad de las decisiones que se toman en el árbol, por lo que este nuevo modelo puede suponer un punto de partida para hacer que los modelos de aprendizaje automático sean más interpretables, lo que puede ser de gran utilidad en campos como la medicina, la sociología, la meteorología, el derecho, etc.

Palabras clave

Aprendizaje automático, inteligencia artificial, red neuronal, árbol de decisión, árbol de decisión de hoja caduca, precisión, interpretabilidad

Abstract

The research and development on machine learning has exploded in recent years due to the revolution of neural networks, which have become the greatest hope in the field of artificial intelligence after the improvements in hardware performance and the availability of high amounts of data. Nevertheless, it appears that advancements have been slowed down recently, probably because neural networks are reaching their precision limits. Moreover, neural networks have an imperfection that has not been remarked frequently: they are a black box in which it is not possible to understand the precision procedure of the network. In consequence, despite their precision, they do not provide us with strong arguments to justify their predictions. Decision trees, on the other hand, are a really interpretable model, though little precise. The objective of the study of this project is the creation of a mixed model between neural networks and decision trees: the deciduous decision tree. This prediction model combines the structure of a decision tree and the predictive capacities of the neural networks, aiming at reaching a custom intermediate point in the dichotomy between precision and interpretability. The results obtained after the analysis of several experiments conducted to the deciduous decision tree reveal that, even with a little loss of precision, the gain of interpretability from the decisions that are made in the tree compensates this loss. Therefore, this new model can be a starting point to make machine learning models be more interpretable, which can be extremely useful in a huge variety of fields, such as medicine, sociology, meteorology, law, etc.

Keywords

Machine learning, artificial intelligence, neural network, decision tree, deciduous decision tree, precision, interpretability

Table of Contents

1 Introduction 1

1.1 Scope . . . 2

1.2 Objectives . . . 3

1.3 Work methodology . . . 3

1.4 Structure of the document . . . 4

2 Basic theory on machine learning 5

2.1 Machine learning basics . . . 5

2.2 Example: linear regression . . . 7

2.3 Neural networks . . . 8

2.4 Decision trees . . . 12

2.5 Precision - interpretability dichotomy . . . 12

3 Technology 13

3.1 Framework: Tensorflow (Python) . . . 13

3.2 IDE: PyCharm . . . 15

4 Deciduous decision tree 17

4.1 Node identification: bit chains . . . 18

4.2 Notation and definitions . . . 19

4.3 Logic unit of a DDTree: the stump . . . 19

4.4 Prediction of the model . . . 22

5 Experimental results 23

5.1 Description and methodology . . . 23

5.2 Experiments . . . 24

5.3 Preliminary experiment: execution time . . . 26

5.4 Experiment results . . . 27

6 Conclusions and future work 29

Bibliography 31

Appendices 33

A Implementation: code 35

A.1 Logic unit: stump . . . 35

A.2 Deciduous decision tree . . . 36

Lists

List of figures

2.1 Example of a linear regression problem 7
2.2 Rosenblatt perceptron 8
2.3 Neural network 9
2.4 Example of a decision tree 12

3.1 PyCharm environment screenshot 15

4.1 DDTree structure example 17
4.2 DDTree with bitchains 18
4.3 Stump 20

5.1 Experiment 1: SNN 25
5.2 Experiment 2: DDTree - 2 leaves 25
5.3 Experiment 3: DDTree - 3 leaves 25
5.4 Experiment 4: DDTree - 4 leaves 25
5.5 Cross-validation accuracy for every experiment 27
5.6 Test results 28

List of tables

5.1 Execution times - MNIST experiment 26
5.2 Test loss - MNIST experiments 28
5.3 Model accuracy - MNIST experiments 28

1 Introduction

Data Science and Artificial Intelligence are two areas in Computer Science that have experienced a huge boost in research and practical applications in recent times. Many companies and research groups have devoted considerable efforts to develop big data and machine learning [1] [2] [3], most of them aiming at becoming data-driven in the near future.

Both fields have many aspects in common. However, there is one factor about machine learning that constitutes a vital dependency from data science: the *learning* in machine learning is considerably better when the data input is high-quality, accurate and representative of reality. Therefore, it is logical that both data science and artificial intelligence are being researched and developed at the same time: improving one field indirectly improves the other, and problems with one affect the other.

In this work we are going to focus on machine learning, specifically on one of the problems that we find in the state of the art of **neural networks**, the most famous model of machine learning. The field of neural networks has experimented a huge development in recent years due to the progress of hardware that has made the emergence of deep learning and deep neural networks possible. These deep neural networks have been proven to be extremely useful and precise for certain problems, such as image or voice recognition, and scientific research has been lately oriented into producing better and better deep learning models, making them quicker and more precise.

However, that development seems to have slowed down as the performance limit of deep learning models has been achieved. While the field's interest exploded between 2017 and 2019, it does not look like there has been further progress in the immediate past and present. The question is: why is that? Why is machine learning development, in particular neural networks and deep learning development, hitting a limit?

In order to answer that question, we are going to explore one characteristic of machine learning models that has been forgotten during the neural network explosion: **interpretability**. The foundation of neural networks is purely based on numerical mathematics, most of which is completely impossible for humans to understand. Neural networks are essentially a black box in which inputs are introduced and from which outputs are extracted, but we can not *understand* what is happening in the process. While it is true that neural networks have proved to be the most precise model in terms of accuracy, that

might not be the best solution in some situations.

For instance, we can imagine a machine learning system that helps doctors diagnose cancer cases based on health data from patients. What is going to be more useful to the doctor: a system that is able to diagnose cancer successfully 95 percent of the times but that the doctor does not understand, or a system that diagnoses successfully less cases but that can be interpreted? In this case, there are many factors to take into account: looking at it from a legal angle, is the doctor responsible for false negatives when they are using a black box system which they can not understand? We can also ask another question: can the doctor intervene in the process so that the false negatives of the system are fixed manually if they do not understand the underlying process? The answers to these questions are not completely conclusive on what is the better scenario, but they definitely show the importance of interpretability in some situations in which machine learning is certainly going to be used in the future.

This analysis leads to another question: are there other machine learning models that are better for interpretability? The answer is, obviously, yes: neural networks are not the only model that has been created, and many other models (regression, naive-Bayes, support vector machines, decision trees, random forests...) may be candidates in our search for interpretability.

We are going to focus on one of them: decision trees. The reason why we focus on decision trees is because there is no other model more *human*: converting input data into output conclusions through individual decisions that are structured hierarchically. If neural networks are the example of high precision but low interpretability, decision trees are the opposite: low precision and high interpretability.

It seems clear that both traits can not be achieved at their best simultaneously. Nonetheless, is there a middle point in which we could maximize precision while demanding a minimum amount of interpretability, or viceversa, using the advantages of both neural networks and decision trees? That is the main objective of this work.

1.1. Scope

This project is going to focus on the design, implementation and analysis of a new machine learning model based on decision trees and other models, specifically neural networks.

The design will provide with the necessary mathematical background through the development of the associated theory, including how the model will perform predictions. For the implementation, we will use the Tensorflow framework in Python language, which is currently one of the standard procedures in the industry. In terms of the analysis, some experiments will be conducted to evaluate the accuracy of the model.

1.2. Objectives

- Set up a work methodology in order to create a strong foundation from which to obtain positive results.
- Describe neural networks and decision trees with their advantages and disadvantages.
- Design a new machine learning model that is flexible, in the sense of being able to prioritize precision or interpretability, and that is based on decision trees.
- Provide a first approach to a possible implementation of that model.
- Analyze the results of that implementation and discuss its usefulness.

1.3. Work methodology

In this section, a brief and structured explanation on the methodology that has been followed to create and develop this project is provided, including the reasoning behind the decisions.

1.3.1. Members of the working team

Due to the nature of this work, which is a degree work, there has only been two members in the working team: the author/student (Fernando Villar) and the advisor/tutor (Alberto Suárez).

1.3.2. Roles

The main role of the **advisor** has been to lead the project, and to check and correct the work done by the author. In addition, the advisor has introduced the theoretical background and the main goals of the project, setting starting points and offering several technologies for the author to choose from.

The main role of the **author** has been to develop the work proposed by the advisor, using the introductory theory and the technology tools provided by the advisor.

1.3.3. Project development phases

The project has been clearly divided in two phases: the theoretical design, and the software development. The theoretical design phase has covered the introductory background explanations to the topic and the design of the new machine learning model; while the software development phase has covered the investigation on which technologies to use, the implementation of the new model using the selected technologies and the analysis and experiments conducted on that implementation.

Theoretical design phase

During this phase, a standard educational methodology has been followed. It has consisted in several meetings between the advisor and the author in which the advisor would present information regarding the scope of the project, explaining the nature and properties of the objects of study and developing the theoretical foundations of the new model to be created.

Software development phase

Once theory and design were determined, software development started. In order to obtain a high quality product, a waterfall paradigm was selected and used as the methodology. In addition, the first phases of a waterfall methodology (conception and analysis) were already covered with the work done in the theoretical design phase, allowing the team to start directly at the (software) design phase.

In this context, the author played the role of the development team and the advisor played the roles of the head of the development team and the end client. The interactions between the author and the advisor can be consequently grouped into two types:

- **Head of the team - team**: After every phase was finished, advisor and author, in the roles of head of the team and development team, had a meeting in which the author showed the development of the product and the advisor decided whether to accept it or suggest changes.
- **End client - team**: At the end of the development, when the implementation was finished, the advisor had to approve the final product depending on the fulfillment of the objectives proposed in the beginning.

1.4. Structure of the document

In chapter 2, we will explain in detail how neural networks and decision trees work, including theoretical development and the mathematics behind them.

In chapter 3, the technology that will be used in order to create the implementation of the new machine learning model will be introduced: the Tensorflow framework.

In chapter 4, the design of the new model will be thoroughly developed, including the mathematics and other details of special relevance.

In chapter 5, some experiments will be conducted using the implementation based on the design of the previous chapter, and the results will be shown and analyzed.

Finally, in chapter 6, some conclusions will be drawn from the entire process, which will lead to some ideas from which to extract starting points for future work on this matter.

2 Basic theory on machine learning

2.1. Machine learning basics

The simplest definition of a **machine learning algorithm** is that it is *an algorithm that is able to learn from data* [4]. More formally, **machine learning** is the study of computer algorithms that improve automatically through experience [5]. These algorithms help us perform tasks that are too difficult to solve with fixed programs written by human beings [4], such as recognizing the silhouette of a face in a photograph or detecting which digits are associated to a handwritten number.

These tasks are said to be “too difficult to solve with fixed programs” because their nature would force any program written to perform them to be extremely large and unmanageable. For instance, let's consider the case of a sound file from which we want to transcribe the words of a human person. How does someone tackle this problem? Which set of rules has to be created so that the custom program is able to recognize the patterns of wave lengths that determine the sound of the human voice? Every human voice is different, there are hundreds of languages with thousands of different words in each of them: the problem is completely unsolvable from a custom written program perspective.

In order to be able to perform these difficult tasks, we need to create a process that helps the algorithm **learn**, and that learning process requires the use of **data**. In the end, what is desired from a machine learning algorithm is to receive an output (silhouette of a face, digits, transcription of words) from an input (pictures, sounds), therefore machine learning tasks are described in terms of how the system should process an example [4]. Using this paradigm, there are many types of tasks, but we are going to focus on two of them:

- **Classification tasks**: In this type of task, we have a set of classes $\{1, 2, ..., k\}$ and we want to know to which class our input belongs to. In consequence, if we let $\mathcal{I}$ be the set of possible inputs, our machine learning algorithm is required to learn a function $f : \mathcal{I} \to \{1, 2, ..., k\}$.
- **Regression tasks**: In this case, what we require from the algorithm is to predict a numerical value from our input data. Therefore, we want the machine learning algorithm to learn a function $f : \mathcal{I} \to \mathbb{R}$.

In general, data is preprocessed so that $\mathcal{I} = \mathbb{R}^n$, where n is the number of attributes of the problem, i.e. the number of different measures that the input data contains. For example, in the digit-handwritten numbers problem, if the images with handwritten numbers have a shape of 28x28 pixels, $n = 28 \cdot 28 = 784$, which means that each pixel is an attribute of the problem.

To evaluate the abilities of a machine learning algorithm, we must design a quantitative measure of its performance, specific to the task being carried out by the system [4]. For classification tasks, the most common measure is the **accuracy** of the model, or equivalently, the **error rate**. This measure is simply the proportion of examples that are correctly (or incorrectly) classified. For regression tasks, another measure must be design, for obvious reasons: it is not about predicting the exact number, but about getting close enough. In this case, a measure like the **mean squared error** is particularly useful. In general, each task requires a different way of measuring success.

In terms of how our algorithms gain *experience*, there are two different groups of algorithms depending on their process of training: **supervised** and **unsupervised** learning algorithms. In supervised learning algorithms, we provide a training set of data with labels, so that the system can compare its predictions with the real results. For example, using the handwritten numbers example, a supervised learning algorithm would use a data set with a collection of photographs along with the digits associated to the handwritten numbers in the images (that means, the photographs (inputs) are **labelled**). On the contrary, unsupervised learning algorithms make use of unlabelled data sets, and are more focused on detecting patterns in the input data rather than establishing a connection between inputs and outputs.

All of the machine learning algorithms and models that are treated in this work belong to the supervised learning algorithms category. Therefore, for every task, we must provide two data sets: a **training set** and a **test set**, both of them including inputs and labels. The test set must not intervene in the training process, and similarly, the training set must not intervene in the performance measuring process; otherwise, both training and validation would be biased towards whichever data we use, and therefore they would not serve their purpose. If we use test data to train the model, the results of the training would be biased towards that data, when all we want from the test data is to evaluate the *real* accuracy of the model.

In general, the training process consists of evaluating a labelled example, obtaining a measure of its performance with respect to the label, and update the model in order to maximize the performance (or to minimize loss/error rates). The model contains a set of parameters (variables) called **weights**, which are used to perform mathematical operations corresponding to the functions $f : \mathcal{I} \to \mathcal{O}$ to be learned by the algorithm: these weights are the part of the model that is updated in every iteration of the process.

2.2. Example: linear regression

Let $T = \{x_i, y_i\}_{i=0}^n$ be a subset of $\mathbb{R}^2$ which contains 2-dimensional points that are thought to be measurements of some phenomena associated to a straight line $\mathcal{L}$ in the plane (as in figure 2.1). From that data set, the desired task is to receive a coordinate in the horizontal axis $x \in \mathbb{R}$ (input) and obtain the corresponding coordinate in the vertical axis $y \in \mathbb{R}$ (output) so that the point $(x, y) \in \mathbb{R}^2$ is in $\mathcal{L}$.

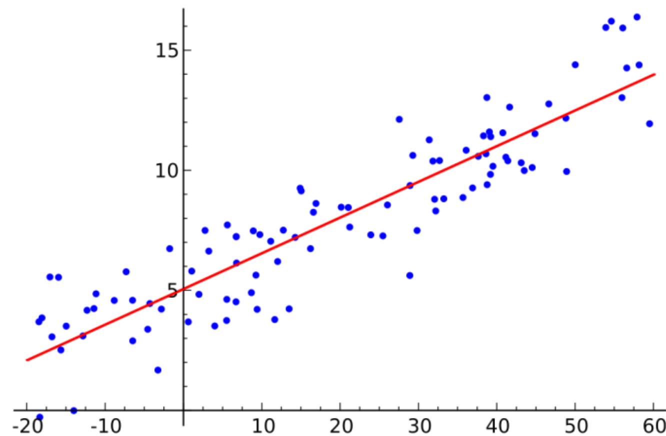


Figure 2.1: Example of a linear regression problem

The straight line $\mathcal{L}$ is described, as any straight line in $\mathbb{R}^2$, with the equation $y = \hat{a}x + \hat{b}$. We need to know the values of $\hat{a} \in \mathbb{R}$ and $\hat{b} \in \mathbb{R}$; therefore, let a and b be the **weights** of our model, so that after the training process we obtain $a = \hat{a}$ and $b = \hat{b}$.

In order to **train** our model, we are going to use T as our training set. As we stated previously, we need a performance measure: in this case, our measure is going to be the training **mean squared error**, that is, the mean of squared differences between the output of our model (that is, given an input x_i, $\hat{y}_i = ax_i + b$) and the real value of the coordinate (label y_i) along the training set:

$$\text{MSE}_{\text{train}} = \frac{1}{n}\sum_{i=0}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n}\sum_{i=0}^{n}(y_i - (ax_i + b))^2 \tag{2.1}$$

Our objective is to find a and b so that the mean squared error is minimal. Therefore, all we need to do is solving an optimization problem for the mean squared error in equation 2.1.

$$\nabla_{(a,b)}\text{MSE}_{\text{train}} = (0, 0)$$

The solutions to this equation are the values of $\hat{a}$ and $\hat{b}$ that we were looking for. In this process, we have updated the weights a and b and we have obtained the new weights $\hat{a}$ and $\hat{b}$ that minimize the error: our model has **learnt** the slope and intercept of $\mathcal{L}$.

2.3. Neural networks

2.3.1. Perceptron

Before explaining what a neural network is, it is convenient to understand the ancestor of all neural networks: the **perceptron**, developed in the 1950s and 1960s by the scientist Frank Rosenblatt [6] [4]. In its simplest definition, a perceptron is a structure that takes several binary inputs x_i and produces a single binary output.

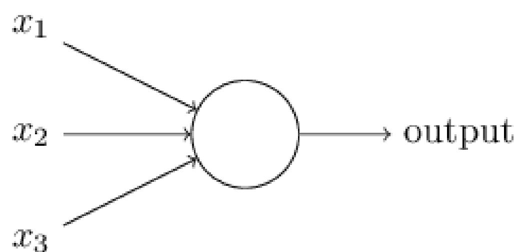


Figure 2.2: Rosenblatt perceptron [6]

Each input is *introduced* in the perceptron along with a weight w_i. In its original definition, the output is determined by the function:

$$\text{output} = \begin{cases} 0, & \text{if } \sum_i w_i x_i \leq \theta \\ 1, & \text{if } \sum_i w_i x_i > \theta \end{cases}$$

Where θ is a threshold that delimits the region where the output is 0 or 1. However, we can create a more complex and robust perceptron by allowing inputs and outputs to be numbers in $\mathbb{R}$, and by using another **activation function** $\sigma : \mathbb{R} \to \mathcal{O}$ that uses the weighted sum of the inputs and transforms it in an output from the set of outputs $\mathcal{O}$. With those changes, our new output looks like this:

$$\text{output} = \sigma\left(\sum_i w_i x_i\right)$$

In the original definition, $\sigma = \mathbb{1}_{\{\sum_i w_i x_i > \theta\}}$. This construction is an algorithm that transforms a vector in $\mathbb{R}^n$ in an output of any kind through the activation function and the use of some weights: therefore, it can be considered as a machine learning algorithm.

2.3.2. Definition

Now that we now what a perceptron is, the **neural network** is easy to define: it is a network of perceptrons and connections between them.

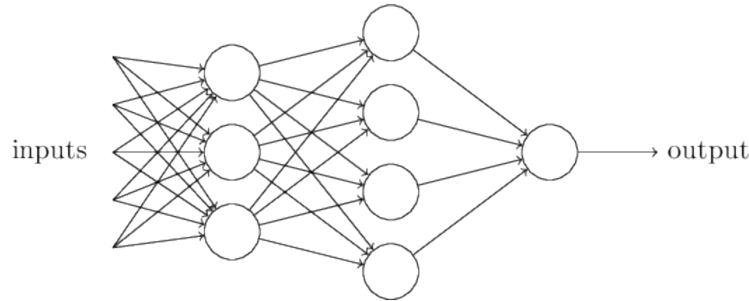


Figure 2.3: Neural network [6]

The network is composed of several **layers**, including the input layer and the output layer. The layers that are located in the middle are called **hidden** layers. In figure 2.3 there is an example of a neural network with 5 inputs, one output and one hidden layer.

The perceptrons that form the neural networks are also called **neurons** in this context (and thus the name neural network). As we can see in the example of figure 2.3, there are 5 inputs, the input layer has 3 neurons, the hidden layer has 4 neurons and the output layer has one neuron. In total, there are $5 \cdot 3 + 3 \cdot 4 + 4 \cdot 1 = 15 + 12 + 4 = 31$ weights in the neural network: for each pair of layers, the number of connections between them is the product of the number of neurons in each layer.

These weights can be represented as a matrix. For each pair of consecutive layers with n and m neurons respectively, the weight matrix is the following, where w_{ij} indicates the weight that connects the j-th neuron of the left layer with the i-th neuron of the right layer:

$$\boldsymbol{W} = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \dots & w_{mn} \end{bmatrix}$$

Therefore, if $\boldsymbol{x} = [x_1, \dots, x_n]$ are the outputs of the left layer, we can calculate the inputs of the right layer $\boldsymbol{y} = [y_1, \dots, y_m]$ by applying the matrix product $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x}$. Let $\boldsymbol{x^k}$ with $k \in \{1, \dots, l\}$ where l is the number of layers of the neural network be the set of vectors containing the inputs of the k-th layer ($\boldsymbol{x^1}$ is the vector of inputs), let $\boldsymbol{W^k}$ be the matrices of weights between the $(k-1)$-th layer and the k-th layer ($\boldsymbol{W^1}$ is the matrix of weights between the inputs and the first layer) and let $\sigma : \mathbb{R} \to \mathbb{R}$ be the activation function. Then, the output $\boldsymbol{y}$ of the model can be calculated with the formula:

$$\boldsymbol{y} = \sigma(\boldsymbol{W^l}\boldsymbol{x^l}) = \sigma(\boldsymbol{W^l}\sigma(\boldsymbol{W^{l-1}}\boldsymbol{x^{l-1}})) = \dots = \sigma(\boldsymbol{W^l}\sigma(\boldsymbol{W^{l-1}}\sigma(\dots\sigma(\boldsymbol{W^2}\sigma(\boldsymbol{W^1}\boldsymbol{x^1}))\dots)))$$

2.3.3. Training

The most standard procedure to train a neural network is a combination of two algorithms: **gradient descent** and **back-propagation**.

Gradient descent

Gradient descent is an iterative optimization algorithm for finding a local minimum of a differentiable function. Let $f : \mathbb{R}^n \to \mathbb{R}$ be a differentiable function, let $\boldsymbol{x_i} = [x_1, \ldots, x_n]$ be a vector in $\mathbb{R}^n$ where $i \in \mathbb{N}$ is the current iteration of the algorithm and let $\gamma \in \mathbb{R}$ be the **learning rate**; gradient descent is a process that consists in:

1.– Calculating $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$

2.– Evaluating ∇f in $\boldsymbol{x_i}$: $\nabla f(\boldsymbol{x_i}) \in \mathbb{R}^n$

3.– Updating the value of the vector using $\boldsymbol{x_{i+1}} = \boldsymbol{x_i} - \gamma \nabla f(\boldsymbol{x_i})$

This algorithm converges into a vector $\boldsymbol{x}$ such that $f(\boldsymbol{x})$ reaches a local minimum. A proof of this result can be found in [7], which includes the original proof made by Cauchy in 1847.

Consider now the output layer of the neural network, using the same notation as in section 2.3.2. The function $\mathcal{L} : \mathbb{R}^n \to \mathbb{R}$ is a **loss function** that maps the outputs of the output layer $\boldsymbol{\sigma^l} = [\sigma_1^l, \ldots, \sigma_m^l]$ (after applying the activation function), where $\sigma_j^l = \sigma(\sum_{i=1}^n w_{ji}^l x_i^l)$, to a measure of the discrepancy between the outputs of the neurons and the real outputs from the labels of the training set; and let $\boldsymbol{\sigma} : \mathbb{R}^n \to \mathbb{R}^n$ be a function such that $\boldsymbol{\sigma}(t_1, \ldots, t_n) = (\sigma(t_1), \ldots, \sigma(t_n))$. Our goal is to calculate the gradient of $\mathcal{L}$ with respect to the weights $\boldsymbol{W^l}$ of the output layer.

We have that $\mathcal{L}(\boldsymbol{\sigma^l}) = \mathcal{L}(\boldsymbol{\sigma}(\boldsymbol{W^l x^l}))$. In order to simplify the notation, we will write $\boldsymbol{z}^l = \boldsymbol{W^l x^l}$ so that $\mathcal{L}(\boldsymbol{\sigma^l}) = \mathcal{L}(\boldsymbol{\sigma}(\boldsymbol{z^l}))$ [8]. In order to calculate the partial derivative $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W^l}}$, we will use the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W^l}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{\sigma^l}} \cdot \frac{\partial \boldsymbol{\sigma^l}}{\partial \boldsymbol{z^l}} \cdot \frac{\partial \boldsymbol{z^l}}{\partial \boldsymbol{W^l}} \tag{2.2}$$

The partial derivative $\dfrac{\partial \boldsymbol{z^l}}{\partial \boldsymbol{W^l}}$ is equal to $\boldsymbol{x^l}$, the inputs of the output layer; but the inputs of the output layer are equal to the outputs of the second-to-last layer, thus $\dfrac{\partial \boldsymbol{z^l}}{\partial \boldsymbol{W^l}} = \boldsymbol{\sigma^{l-1}}$. The other two partial derivatives depend on the functions $\mathcal{L}$ and σ, which are usually functions with easy-calculating derivatives, and common for all layers. We give a special name to the product of those terms: $\delta^l = \dfrac{\partial \mathcal{L}}{\partial \boldsymbol{\sigma^l}} \cdot \dfrac{\partial \boldsymbol{\sigma^l}}{\partial \boldsymbol{z^l}}$. Finally, all that is left in order to update the weights $\boldsymbol{W^l}$ is to apply the gradient descent algorithm.

Back-propagation

The back-propagation algorithm is the missing piece that we need in order to update the weights of all the layers in the neural network, and not only the weights of the output layer, as we did in the previous section. It is based on the notion that $\delta^l = \dfrac{\partial \mathcal{L}}{\partial \boldsymbol{z^l}}$ contains the losses associated to each neuron of the layer l, as it is the rate of change of the loss function when we change the input of the activation function.

Applying the chain rule as in equation 2.2 for the weights of the $(l-1)$-th layer, we obtain [8]:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W^{l-1}}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{\sigma^l}} \cdot \frac{\partial \boldsymbol{\sigma^l}}{\partial \boldsymbol{z^l}} \cdot \frac{\partial \boldsymbol{z^l}}{\partial \boldsymbol{\sigma^{l-1}}} \cdot \frac{\partial \boldsymbol{\sigma^{l-1}}}{\partial \boldsymbol{z^{l-1}}} \cdot \frac{\partial \boldsymbol{z^{l-1}}}{\partial \boldsymbol{W^{l-1}}} \tag{2.3}$$

From that product, we already know two terms: δ^l and the derivative of the activation function (which is the same independently of the layer). Besides, the last derivative, as we saw in the previous section, is equal to the outputs of the previous layer $\boldsymbol{\sigma^{l-2}}$; the only term that we need to calculate is $\frac{\partial \boldsymbol{z^l}}{\partial \boldsymbol{\sigma^{l-1}}}$, which is equal to the weights of the layer l: $\boldsymbol{W^l}$. Therefore, the gradient that we are looking for is

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W^{l-1}}} = \delta^l \boldsymbol{W^l} \frac{\partial \boldsymbol{\sigma^{l-1}}}{\partial \boldsymbol{z^{l-1}}} \boldsymbol{\sigma^{l-2}}$$

and we can define again

$$\delta^{l-1} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z^{l-1}}} = \boldsymbol{W^l} \delta^l \frac{\partial \boldsymbol{\sigma^{l-1}}}{\partial \boldsymbol{z^{l-1}}}.$$

We can repeat this process backwards until we reach the input layer using the expressions:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W^k}} = \delta^k \boldsymbol{\sigma^{k-1}}$$

which computes all of the necessary calculations with extreme efficiency.

The combination of gradient descent and back-propagation is what provides neural networks with an efficient iterative training algorithm, which optimizes the weights until the loss function reaches a local minimum (that would hopefully be either the absolute minimum or a small enough local minimum).

2.4. Decision trees

A **decision tree** is another machine learning algorithm that consists of a tree-based structure which contains decisions in the internal nodes and outputs in the leaves. Based on the inputs, decisions are made in the internal nodes to select which path to follow, and the final prediction is the value of the leaf located at the end of the path. The details on how decision trees are trained are out of the scope of this project, because in following chapters a new model which is based on decision trees is thoroughly explained.

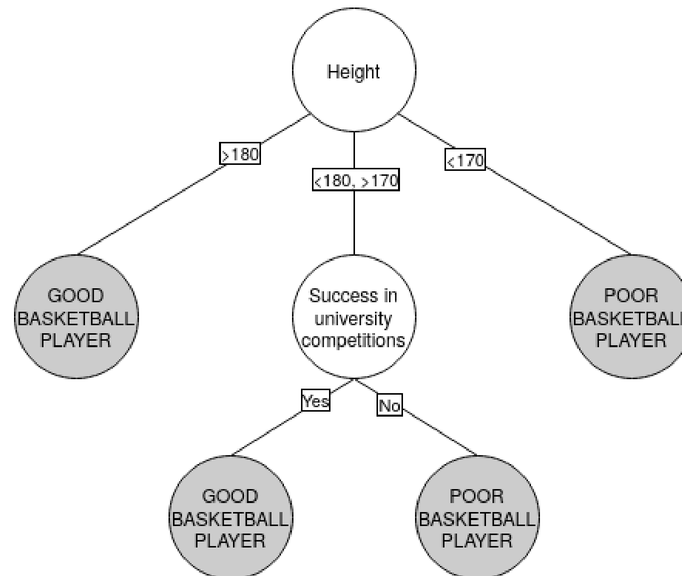


Figure 2.4: Example of a decision tree

2.5. Precision - interpretability dichotomy

What is important for our work is the conceptual difference between neural networks and decision trees. While the former have proven to be really precise when measuring accuracy, the intricacies of how they predict make them really difficult to understand; however, the latter are really easy to understand but have never managed to reach high levels of precision.

It would be ideal to have a model with both traits: **precision** and **interpretability**, but experience has shown that it is extremely difficult to obtain both at the same time. In fact, neural networks seem to be the pinnacle of precision, while decision trees are as interpretable as a model can be. The question is: is there a middle-ground model that gives up a little bit of one trait in exchange of an improvement in the other, and that lets us choose in which degree we want to obtain both of them?

In this work we propose a new model, the **deciduous decision tree**, that aims at answering the previous question with a “yes”.

3 Technology

3.1. Framework: Tensorflow (Python)

The implementation of the model that has been introduced in the previous section is going to be performed in the Tensorflow framework [9], using the Python language. Tensorflow is an open source library for machine learning, and it has a huge variety of resources that allow developers to easily build machine learning applications.

Why using Tensorflow and Python? In terms of the language, the decision was unquestionable, because Python is the standard for machine learning in the present. Python has the advantage of being an object-oriented programming language and really fast and easy to use, besides having numerous libraries that provide interfaces for executing instructions with high efficiency and performance (such as NumPy or Scikit-Learn).

Once Python was the choice, there was only one binary decision left: whether to use Tensorflow (Google) or PyTorch (Facebook). Both libraries are really similar after the 2019 update of Tensorflow 2.0, and the deciding factor was the GPU support that Tensorflow offers for Nvidia GPU users, allowing us to execute our programs faster thanks to the power of parallelism in the GPU.

3.1.1. Examples of neural networks

In the following codes there are two examples of neural networks built in Tensorflow.

In the first example, a neural network with one input layer, one hidden layer and one output layer is built with the “RMSProp” optimizer (which generates a training process similar to the combination of gradient descent and back-propagation, but a little more refined) and the “categorical cross-entropy” loss function. The “fit” instruction is the one that performs training.

In the second example [10], the neural network created is similar, but with another loss function, the mean squared error (in this example, the task is regression). In this case, there is another algorithm in the “for” loop: cross validation, so that loss and accuracy are calculated with even more precision.

Code 3.1: Example of a neural network definition and training in Tensorflow (multi-class classification)

```
1  model = models.Sequential()
2  model.add(layers.Dense(32, activation='relu', input_shape=(784,)))
3  model.add(layers.Dense(32, activation='relu'))
4  model.add(layers.Dense(10, activation='softmax'))
5  model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
6
7  history = model.fit(x_train, y_train, epochs=15, batch_size=64, validation_data=(x_test, y_test))
```

Code 3.2: Example of a neural network definition and training in Tensorflow (regression)

```
1  def build_model():
2      model = models.Sequential()
3      model.add(layers.Dense(64, activation='relu', input_shape=(train_data.shape[1],)))
4      model.add(layers.Dense(64, activation='relu'))
5      model.add(layers.Dense(1))
6      model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
7      return model
8
9  k = 4
10 num_val_samples = len(train_data) // k
11 num_epochs = 100
12 all_scores = []
13
14 for i in range(k):
15     print('processing fold #', i)
16     val_data = train_data[i *num_val_samples: (i + 1) *num_val_samples]
17     val_targets = train_targets[i *num_val_samples: (i + 1) *num_val_samples]
18
19     partial_train_data = np.concatenate([train_data[:i *num_val_samples], train_data[(i + 1) *
           num_val_samples:]], axis=0)
20     partial_train_targets = np.concatenate([train_targets[:i *num_val_samples], train_targets[(i + 1) *
           num_val_samples:]], axis=0)
21     model = build_model()
22
23     model.fit(partial_train_data, partial_train_targets, epochs=num_epochs, batch_size=1, verbose=0)
24     val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
25     all_scores.append(val_mae)
```

3.2. IDE: PyCharm

Among the possible choices for the numerous IDEs compatible with Python, PyCharm was selected because of the easiness at working with virtual environments. When working in Python, being able to work with several virtual environments becomes a crucial factor in developing applications, as it allows the developer to switch between different versions of the libraries that are used.

Besides, PyCharm also provides with a simple and efficient debugging system, which has proven to be quite helpful when dealing with programming inaccuracies.

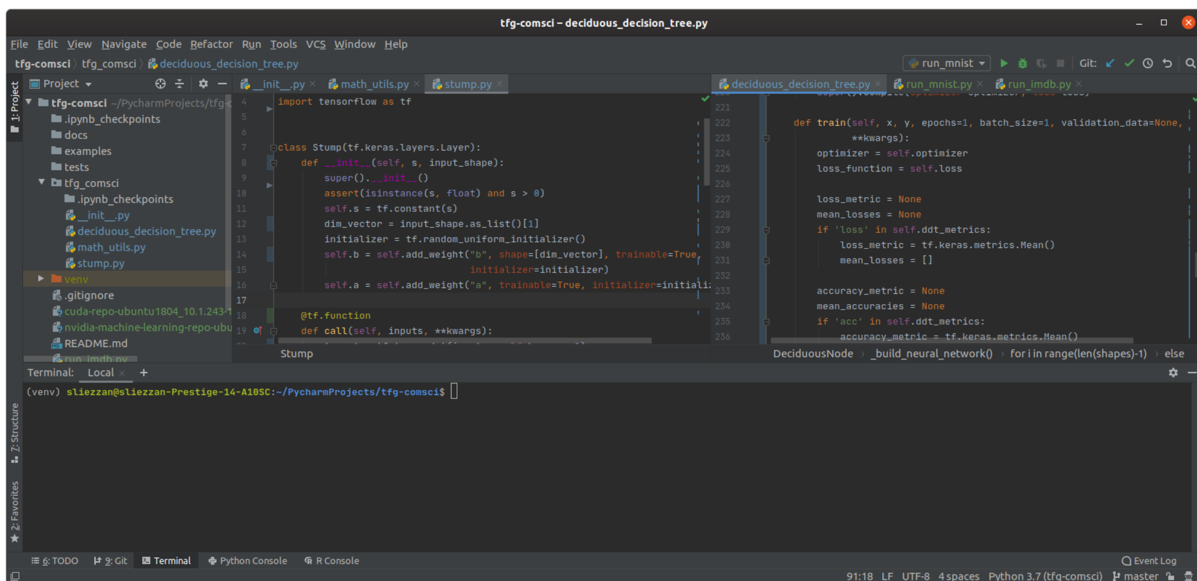


Figure 3.1: PyCharm environment screenshot

4 Deciduous decision tree

A **deciduous decision tree** (which will be addressed in the future as a DDTree) is defined as a prediction model which combines a decision tree with other prediction models by changing the content of the leaves of the decision tree: instead of containing predictions, these leaves contain different prediction models, from which the final prediction is extracted. These prediction models can be of any nature: neural networks, support vector machines, etc.

The focus of our work is to study binary deciduous decision trees in depth. The word “binary” indicates that the underlying tree is a binary decision tree. An example of a binary deciduous decision tree is shown in figure 4.1, which contains, in this case, three leaves (neural networks) and two decision nodes.

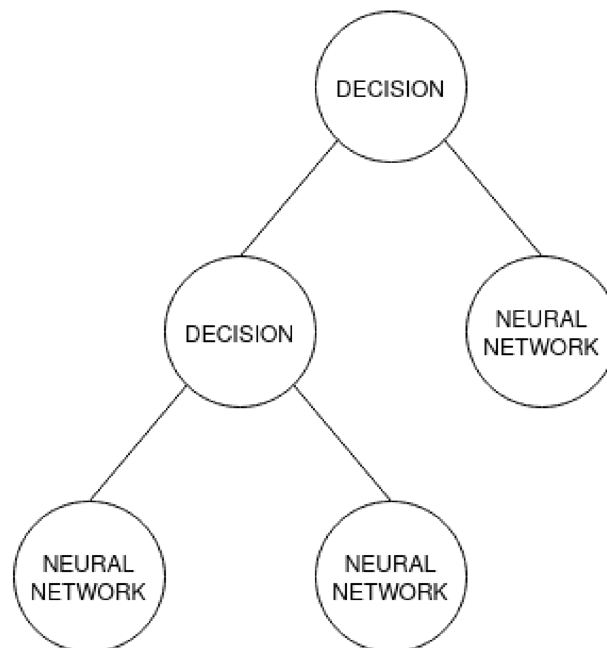


Figure 4.1: Example of a DDTree

4.1. Node identification: bit chains

The first task that requires development is the creation of some notation, definitions and a system to identify the nodes of the DDTree. In order to accomplish the latter objective, we will use **bit chains**, that is, sets of ones and zeros that identify each node of the tree uniquely. Our goal, in this case, is to define the position of each node from the position of its ancestors, using the decisions taken along the path from the root node: essentially, whether we choose left or we choose right from the root node until we arrive at the node we want to identify.

We will use the convention of identifying a left choice with a 0, and a right choice with a 1. Therefore, the length of the bit chain associated to a node will be equal to its depth, and will contain all of the binary decisions required to go from the root node to that node. For instance, in figure 4.2, the leftmost leaf of the tree is identified by the bit chain $[0, 0]$, as the path that goes from the root node to this leaf requires two left choices (and has depth 2); while the rightmost leaf of the tree is identified by the bit chain $[1]$, as the path that goes from the root node to this leaf requires one right choice (and has depth 1).

A bit from these chains is a $c^i \in \{0, 1\}$ where i stands for the depth of the arrival node after making the choice. Consequently, using the previous example, we would have $c_1 = 0$ and $c_2 = 0$. The bit chain associated to the root node is the empty set $[\,]$.

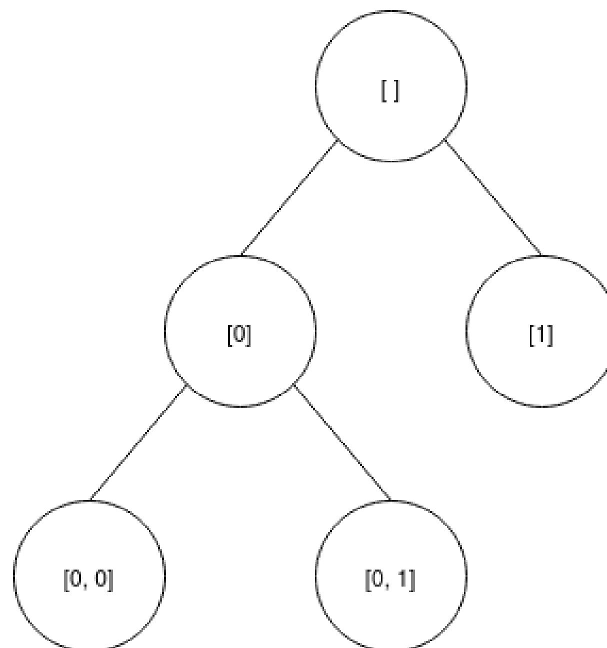


Figure 4.2: Tree nodes identification through bit chains

4.2. Notation and definitions

We define $\mathcal{N}$ as the set of all nodes in the tree. The elements of $\mathcal{N}$ will be indicated by the letter η and will be identified by the bit chain $\boldsymbol{c_\eta} = [c_\eta^1, c_\eta^2, ..., c_\eta^h]$, where h is the depth of the node η.

Besides, we define $\mathcal{L}$ as the set of all leaves in the tree. Its elements will also be indicated by the letter η and will follow the same bit chain identification system, as they are nodes as well. Logically, $\mathcal{L} \subset \mathcal{N}$.

It will be extremely useful for us to identify nodes with bit subchains, obtained by slicing bit chains associated to descendant nodes: for example, to access the parent of a certain node, we can take the bit chain $[c_\eta^1, c_\eta^2, ..., c_\eta^{h-1}]$ obtained from $\boldsymbol{c_\eta}$. Therefore, given a node $\eta \in \mathcal{N}$, we identify any of its ancestors with $\boldsymbol{c_\eta^d} = [c_\eta^1, c_\eta^2, ..., c_\eta^d]$, where d is the depth of the corresponding ancestor (obviously, $0 \leq d < h$). For every tree and every node $\eta \in \mathcal{N}$, the root node of the tree can be represented as $\boldsymbol{c_\eta^0} = [\,]$.

The advantage of using this kind of notation is that we can define any node of the tree by selecting a descendant leaf, and then slice its associated bit chain until reaching the desired depth. This will be useful for recursive definitions that will arise naturally from the tree structure.

4.3. Logic unit of a DDTree: the stump

Prediction models such as neural networks are built with logic units called layers (input layer, output layer and hidden layers, for example). In our case, the logic unit of a DDTree is the **stump**, a 1-depth binary tree from which we exclude its root node. Despite excluding this root node from the structure of a stump, we will keep referring to it as the root node of the stumpïn order to simplify the language.

If we look closer at the structure of DDTrees (figures 4.1 and 4.2), we can see they can be completely built with the root node of the tree and stumps connected appropriately. In figure 4.3 there is an example of a stump, where the lack of root node is illustrated.

In every stump, the root node is a decision node, where the choice between left and right node must be made. To make this decision we will use a logistic function instead of a standard boolean test. In consequence, a fuzzy tree will be created, in which each input is not associated to the prediction of a leaf of the tree; it is associated with a set of **membership degrees**, which are the weights of the predictions of the leaves in the final prediction. This final prediction will be the weighted sum of the predictions of the leaves, using their membership degrees. This will be really beneficial to train our model, as the logistic function is continuous and differentiable (unlike boolean tests), and its derivative is easy to calculate.

4.3.1. Stumps with leaves

We will start our work on stumps with a particular kind of stumps as shown in figure 4.3, that is, those stumps whose children are leaves of the tree (and, in consequence, contain different prediction models; in this example, neural networks).

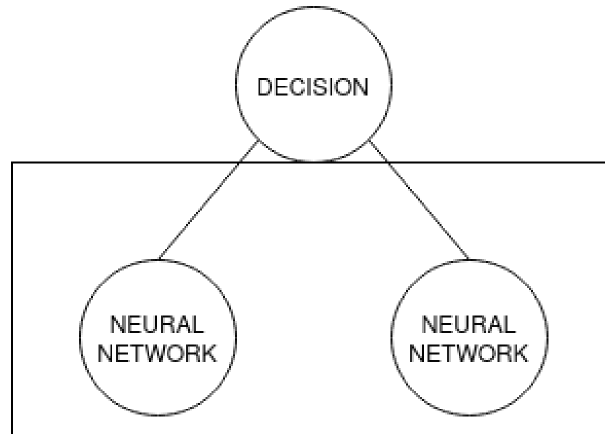


Figure 4.3: Stump

Let η be the root node of the stump and $\boldsymbol{c_\eta}$ its associated bit chain, and let η_L and η_R be its left and right children respectively, with bit chains $\boldsymbol{c_{\eta_L}}$ y $\boldsymbol{c_{\eta_R}}$. If D is the number of features of the problem (we will also call it the **dimension**), $\boldsymbol{x} \in \mathbb{R}^D$ is the input vector, $\boldsymbol{b_{c_\eta}} \in \mathbb{R}^D$ is a weights vector with the restriction $||\boldsymbol{b_{c_\eta}}|| = 1$, $a_{\boldsymbol{c_\eta}} \in \mathbb{R}$ is the bias and $s_{\boldsymbol{c_\eta}} \in \mathbb{R}$ is a width parameter such that $s_{\boldsymbol{c_\eta}} > 0$ (these last three parameters depend on the stump we are in inside the tree), the membership degree of the right child is obtained from the following logistic function:

$$\mu_{\eta_R}(\boldsymbol{x}; \boldsymbol{b_{c_\eta}}, a_{\boldsymbol{c_\eta}}, s_{\boldsymbol{c_\eta}}) = \frac{1}{1 + e^{-\frac{1}{s_{\boldsymbol{c_\eta}}}(\boldsymbol{b}_{\boldsymbol{c_\eta}}^T \boldsymbol{x} - a_{\boldsymbol{c_\eta}})}}$$

And the membership degree of the left child is defined as:

$$\mu_{\eta_L}(\boldsymbol{x}; \boldsymbol{b_{c_\eta}}, a_{\boldsymbol{c_\eta}}, s_{\boldsymbol{c_\eta}}) = 1 - \mu_{\eta_R}(\boldsymbol{x}; \boldsymbol{b_{c_\eta}}, a_{\boldsymbol{c_\eta}}, s_{\boldsymbol{c_\eta}})$$

Both membership degrees satisfy $0 \leq \mu_i(\boldsymbol{x}; \boldsymbol{b_{c_\eta}}, a_{\boldsymbol{c_\eta}}, s_{\boldsymbol{c_\eta}}) \leq 1,\ i \in \{\eta_L, \eta_R\}$.

Regarding the different prediction models that are contained in the children, let their weights (trainable parameters) be $\boldsymbol{\omega}^{[i]},\ i \in \{\eta_L, \eta_R\}$. Then, we express the predictions of these models through the functions:

$$o_i(\boldsymbol{x}; \boldsymbol{\omega}^{[i]}), \quad i \in \{\eta_L, \eta_R\}$$

Therefore, the final prediction of the stump is the output of the following function, whose parameters are all trainable with the exception of s (which will be dealt with independently):

$$o(\boldsymbol{x};\boldsymbol{b}_{\boldsymbol{c}_\eta},a_{\boldsymbol{c}_\eta},s_{\boldsymbol{c}_\eta},\boldsymbol{\omega}^{[\eta_L]},\boldsymbol{\omega}^{[\eta_R]}) = \mu_{\eta_L}(\boldsymbol{x};\boldsymbol{b}_{\boldsymbol{c}_\eta},a_{\boldsymbol{c}_\eta},s_{\boldsymbol{c}_\eta})o_{\eta_L}(\boldsymbol{x};\boldsymbol{\omega}^{[\eta_L]}) + \mu_{\eta_R}(\boldsymbol{x};\boldsymbol{b}_{\boldsymbol{c}_\eta},a_{\boldsymbol{c}_\eta},s_{\boldsymbol{c}_\eta})o_{\eta_R}(\boldsymbol{x};\boldsymbol{\omega}^{[\eta_R]})$$

4.3.2. Generalized stumps

We will finish our work on stumps considering those stumps whose children are not leaves, but other internal decision nodes. In the previous section we worked with the parameters $\boldsymbol{b}_{\boldsymbol{c}}$, $a_{\boldsymbol{c}}$ and $s_{\boldsymbol{c}}$ for a stump identified with a bit chain $\boldsymbol{c}$. In order to simplify the language, and considering any node of the tree with bit chain $\boldsymbol{c}^d_{\boldsymbol{\eta}}$, we will encapsulate all of these paremters into the vector:

$$\boldsymbol{\theta}_{\boldsymbol{c}^d_\eta} = (\boldsymbol{b}_{\boldsymbol{c}^d_\eta}, a_{\boldsymbol{c}^d_\eta}, s_{\boldsymbol{c}^d_\eta})^T$$

Regarding other prediction models weights of the leaves, in case the stump contains them, they will follow the same criteria, and will receive the notation $\boldsymbol{\omega}^{[c_\eta]}$. We do not use the upper index d in this case because these nodes are always leaves.

The membership degrees of the children nodes in the stump remain the same: if $\boldsymbol{c}_{\boldsymbol{\eta_L}}$ and $\boldsymbol{c}_{\boldsymbol{\eta_R}}$ are the bit chains of the children of the node with bit chain $\boldsymbol{c}^d_{\boldsymbol{\eta}}$, then:

$$\mu_{\boldsymbol{c}_{\eta_R}}(\boldsymbol{x};\boldsymbol{\theta}_{\boldsymbol{c}^d_\eta}) = \frac{1}{1+\exp(-s^{-1}_{\boldsymbol{c}^d_\eta}(\boldsymbol{b}^T_{\boldsymbol{c}^d_\eta}\boldsymbol{x} - a_{\boldsymbol{c}^d_\eta}))}, \qquad \mu_{\boldsymbol{c}_{\eta_L}}(\boldsymbol{x};\boldsymbol{\theta}_{\boldsymbol{c}^d_\eta}) = 1 - \mu_{\boldsymbol{c}_{\eta_R}}(\boldsymbol{x};\boldsymbol{\theta}_{\boldsymbol{c}^d_\eta})$$

However, the root node of the stump has another membership degree with respect to its parent; and the pattern continues until the root node of the tree. We defined the **global membership degree** as the membership degree with respect to the tree node, and we denote it as $M(\boldsymbol{x};\boldsymbol{c}^d_{\boldsymbol{\eta}})$. Therefore, the global membership degree of the root node of the stump is $M(\boldsymbol{x};\boldsymbol{c}^d_{\boldsymbol{\eta}})$, while the global membership degrees of the children are

$$M(\boldsymbol{x};\boldsymbol{c}_{\boldsymbol{\eta_R}}) = \mu_{\boldsymbol{c}_{\eta_R}}(\boldsymbol{x};\boldsymbol{\theta}_{\boldsymbol{c}^d_\eta})M(\boldsymbol{x};\boldsymbol{c}^d_{\boldsymbol{\eta}})$$

and

$$M(\boldsymbol{x};\boldsymbol{c}_{\eta_L}) = \mu_{\boldsymbol{c}_{\eta_L}}(\boldsymbol{x};\boldsymbol{\theta}_{\boldsymbol{c}^d_\eta})M(\boldsymbol{x};\boldsymbol{c}^d_{\boldsymbol{\eta}}) = M(\boldsymbol{x};\boldsymbol{c}^d_{\boldsymbol{\eta}}) - M(\boldsymbol{x};\boldsymbol{c}_{\eta_R}).$$

At this point, we have defined all of the objects necessary to calculate the global membership degree of any node from the local membership degrees of itself and all of its ancestors.

Let $\boldsymbol{c_\eta}$ be the bit chain of a node $\eta \in \mathcal{N}$ of depth h. Then, its global membership degree is:

$$M(\boldsymbol{x}; \boldsymbol{c_\eta}) = \prod_{d=0}^{h} \mu_{\boldsymbol{c}_\eta^d}(\boldsymbol{x}; \boldsymbol{\theta}_{\boldsymbol{c}_\eta^d})$$

where the membership degree of the root node is $\mu_{\boldsymbol{c}_\eta^0}(\boldsymbol{x}; \boldsymbol{\theta}_{\boldsymbol{c}_\eta^0}) = 1$.

4.4. Prediction of the model

After all of the previous work, only two parameters and some values are left to be defined. The parameters are:

$$\boldsymbol{\theta} = \{\boldsymbol{\theta}_{\boldsymbol{c}_\eta}\}_{\eta \in \mathcal{N} \setminus \mathcal{L}}$$

$$\boldsymbol{\omega} = \{\boldsymbol{\omega}^{[\boldsymbol{c}_\eta]}\}_{\eta \in \mathcal{L}}$$

which we will use to simplify notation, and are useful to encapsulate all of the parameters of every node (or leaf) explained previously. The values are:

$$o_{\boldsymbol{c}_\eta}(\boldsymbol{x}; \boldsymbol{\omega}^{[\boldsymbol{c}_\eta]})$$

which correspond to the output of the prediction models contained in the leaves of the tree $\eta \in \mathcal{L}$.

In consequence, the prediction of our model ends up being:

$$o(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{\omega}) = \sum_{\eta \in \mathcal{L}} M(\boldsymbol{x}; \boldsymbol{c_\eta}) o_{\boldsymbol{c}_\eta}(\boldsymbol{x}; \boldsymbol{\omega}^{[\boldsymbol{c}_\eta]}) = \sum_{\eta \in \mathcal{L}} \left(o_{\boldsymbol{c}_\eta}(\boldsymbol{x}; \boldsymbol{\omega}^{[\boldsymbol{c}_\eta]}) \prod_{d=0}^{h} \mu_{\boldsymbol{c}_\eta^d}(\boldsymbol{x}; \boldsymbol{\theta}_{\boldsymbol{c}_\eta^d}) \right)$$

5

Experimental results

In this chapter, we are going to conduct several experiments in order to evaluate our new prediction model, the **binary deciduous decision tree**. A basic implementation of the DDTree is shown in Appendix A, which incorporates some details that are explained below.

5.1. Description and methodology

The prediction model used in the leaves of the binary deciduous decision tree is a neural network. In this case, all of the neural networks in each tree have the same structure (number of layers and neurons in each layer). Training is performed using the standard procedure of combining **back-propagation** and **gradient descent**. This part of the process is controlled by Tensorflow, as this library contains tools for automatic gradient calculation and update.

Each problem has its own associated loss function. In this experiment, we are going to use the **MNIST database** (http://yann.lecun.com/exdb/mnist/), which is a **multi-class** classification problem for which the **categorical cross-entropy** will be used as loss function. By default, the optimizer used is the “RMSProp”, which is also already implemented in the Tensorflow library. Other optimizers could be used, such as the “SGD” (stochastic gradient descent). Learning rate is defaulted to the value $0{,}01$.

The MNIST database contains 70,000 samples of images with handwritten numbers from 0 to 9, and the problem consists in predicting a number from a handwritten number. The input data consists of 2-D square matrices of dimension 28, the size of the scanned images with numbers handwritten. The 70,000 samples are split into a **training set** of 60,000 samples and a **test set** of 10,000 examples. The prediction model will generate probability vectors of length 10 using the `softmax` activation function in the last layer of the neural networks located at the leaves of the tree.

The only non-trainable parameter of the model, the value s, will be set to $s = 0{,}5$. This value controls the width of the logistic function assigned to every decision node. This work does not include a specific analysis on this value due to computational limitations, but it would be of great interest to analyze how the model behaves when the logistic function approaches a boolean test (that is, when the logisitic function is really narrow).

5.2. Experiments

Four different experiments will be conducted on the MNIST database using four types of binary deciduous decision trees. Every neural network in these DDTrees contains one hidden layer with a varying number of neurons; the number of neurons of the hidden layer will be the hyperparameter of the problem.

Firstly, for each experiment, **k-fold cross validation** will be performed to obtain the best value for the hyperparameter, with $k = 5$ and the hyperparameter varying between 10, 20, 30, 40 and 50. This means that, for each value of the hyperparameter, the model will be trained independently 5 times: for each training process, the training set will be divided in two subsets, one of them used for pure training (containing $4/5$ of the samples - 800) and the other used for calculating loss and accuracy on an independent data set (containing $1/5$ of the samples - 200). The former set is the **train set** and the latter set is the **validation set**, for which we will calculate loss and accuracy in each step of the training loop.

As we are interested in the behaviour of the models depending on the number of neurons in the hidden layer, the training loop is going to be repeated 30 times for the SNN and 15 times for the DTrees (due to computation limitations): we will observe that, although models may not converge into their best possible accuracy, it will be clear which hyperparameter is the optimal one.

Once we have found the optimal hyperparameter, our goal is to find the number of iterations of the training loop for which validation loss is minimum (or, equivalently, validation accuracy is maximum), and then train our models with the optimal number of neurons in the hidden layer, for the optimal number of training loop iterations. Due to computation limitations, in this step only 10,000 samples from the MNIST Database will be used for training.

In order to perform this task, we will train the models for a high number of epochs (50 epochs) and then analyze when they start to overfit, that means, when validation loss starts increasing instead of decreasing. The optimal number of epochs for each model will be the number of epochs before validation loss starts increasing.

Finally, using the **test set** that is provided by the MNIST database, we will obtain the real accuracy of the models.

The DDTree structures are:

- A **standard neural network (SNN)** (DDTree of depth 0).
- A **deciduous decision tree of 2 leaves (DDTree-2)**.
- A **deciduous decision tree of 3 leaves (DDTree-3)**.
- A **deciduous decision tree of 4 leaves (DDTree-4)**.

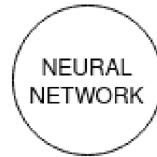


Figure 5.1: Experiment 1: SNN

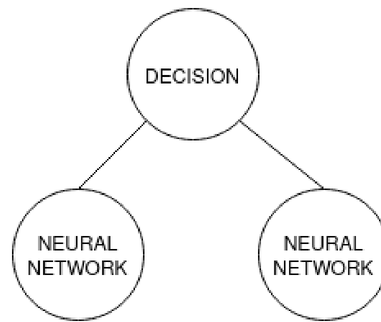


Figure 5.2: Experiment 2: DDTree - 2 leaves

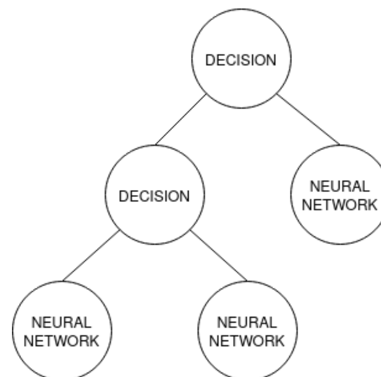


Figure 5.3: Experiment 3: DDTree - 3 leaves

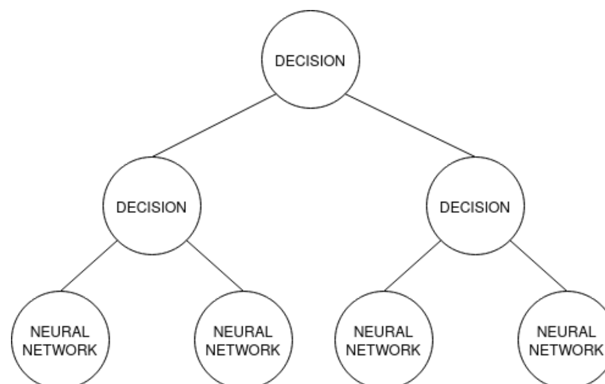


Figure 5.4: Experiment 4: DDTree - 4 leaves

5.3. Preliminary experiment: execution time

Apart from measuring the precision of different deciduous decision trees, it is also interesting to measure how long it takes for the DDTree to perform the training process. In order to analyze execution time, we will compare the execution time of the training process for a standard neural network (SNN) and for the deciduous decision tree of figure 4.1. This experiment is run using the mechanisms provided by the Tensorflow-GPU module, which essentially optimizes calculations by using a Nvidia GPU.

The results are shown in table 5.1. The conclusion that we can extract is that the DDTree is considerably slower. The processing of a single sample takes approximately 12 times longer for the DDTree, probably because our implementation is not as optimized as the Tensorflow libraries. In fact, the design of the library allows for great optimization of matrix calculations, which is what SNNs require; our implementation is based on a tree structure, which implies that not only matrix operations but also other non-optimized tasks that significantly slow down the whole process are performed.

Process	Mean time (DDTree)	Mean time (SNN)
Training - one sample processing	0.2ms	0.013ms
Training loop (1 epoch)	12.8s	1s
Training (15 epochs)	3.2 min	15s

Table 5.1: Execution times - MNIST experiment

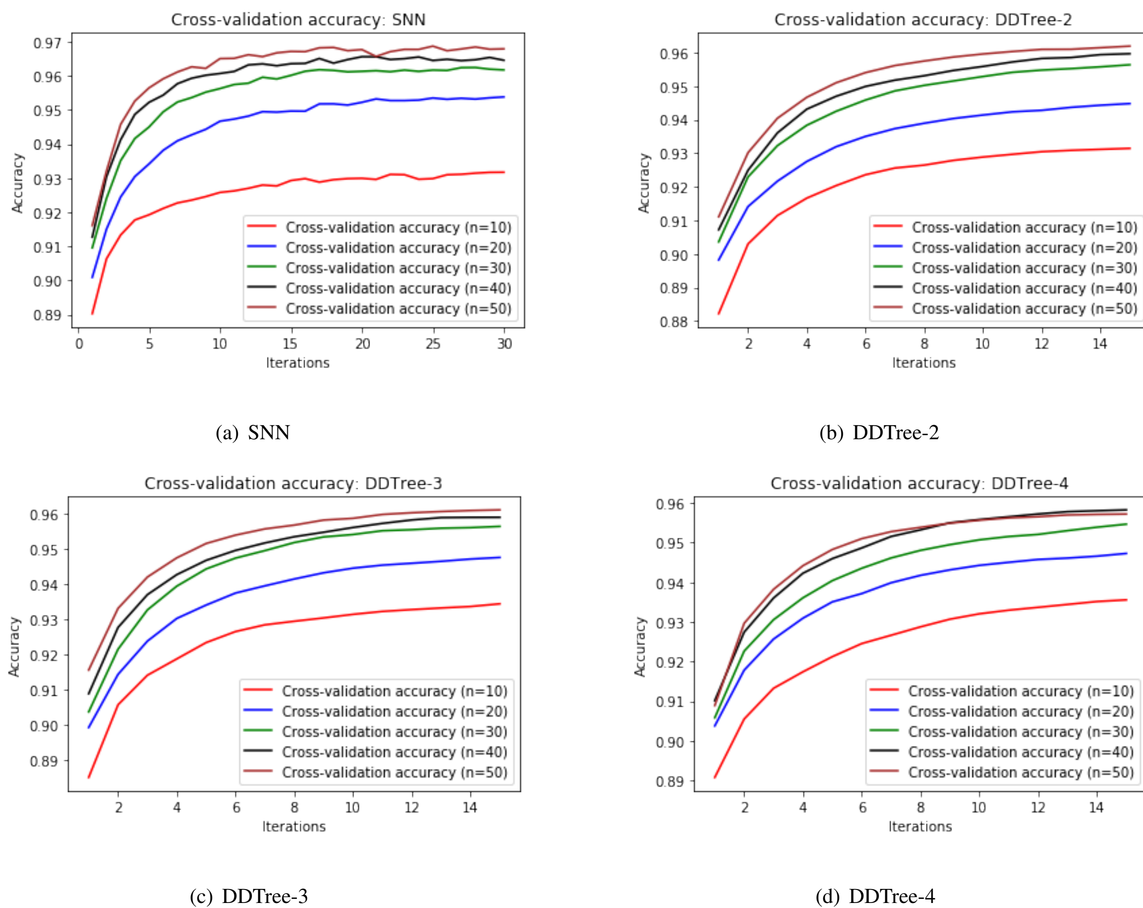
An interesting lesson that we can learn from analyzing execution time results is that there might be another approach to the problem (in terms of implementation) that could improve performance significantly, which is to consider a tree as a set of sets of nodes containing paths from the root node to the leaves, instead of a recursive structure. For instance, a tree like the one shown in figure 4.2 could be redefined as the following set of paths:

$$[([\,],[0],[0,0]),([\,],[0],[0,1]),([\,],[1])]$$

Which can be transformed into a matrix whose rows contain all of the necessary information for each path. There is definitely room to improve performance in this direction, and therefore the huge discrepancy between our DDTree and a SNN in terms of execution time is something that could be improved and should not be worrying.

5.4. Experiment results

The results of the first part of the experiments are shown in figure 5.5. It is clear that the optimal results are achieved with $n = 50$ neurons in the hidden layer. Besides, we can see that the higher this number, the smaller the improvement. Therefore, we will use $n = 50$ from now on, although there might be an even better value for that hyperparameter.



(a) SNN

(b) DDTree-2

(c) DDTree-3

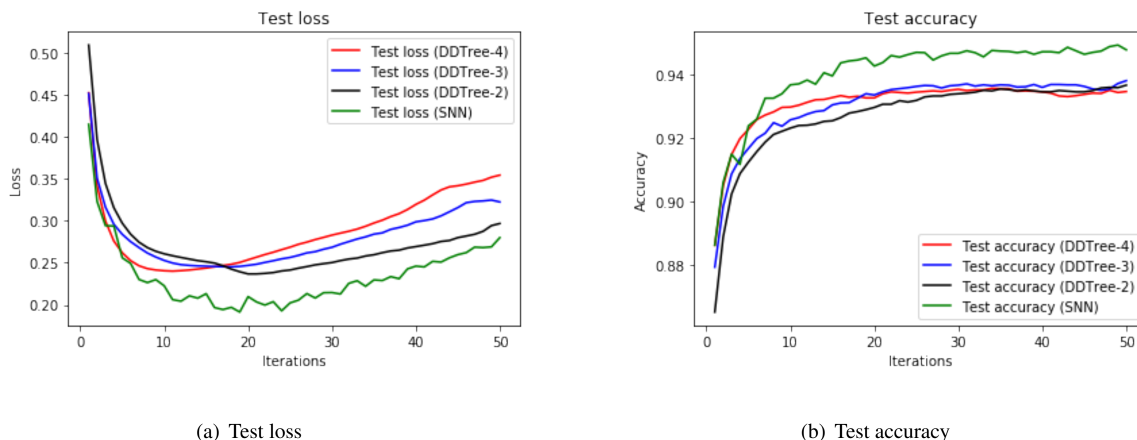
(d) DDTree-4

Figure 5.5: Cross-validation accuracy for every experiment

Now that we have selected the value of our hyperparameter, we will train all models for 50 epochs in order to know which is the optimal number of training loops for each of them. These results are shown in figure 5.6 and table 5.2 (which highlights the optimal number of epochs for each model).

We can observe that the results are quite similar: however, the SNN seems to have been more accurate than the DDTrees, while in theory a SNN can be fit into a DDTree if all of the leaves contain that exact same SNN, which gives DDTree more degrees of freedom and therefore it should have better precision. These results are most likely obtained because of computational limitations and the relatively short amount of samples used as a consequence, but it is also possible that more complex structures like the DDTrees are less likely to find the absolute minimum of the loss function.

In order to confirm these conclusions, we need to calculate the real precision of these models when dealing with new data, using the optimal number of epochs. The results are shown in table 5.3.



(a) Test loss

(b) Test accuracy

Figure 5.6: Test results

Epochs | Model	DDTree-4	DDTree-3	DDTree-2	SNN
9	0.2406	0.2528	0.2607	0.2226
10	**0.2401**	0.2497	0.2584	0.2061
11	0.2407	0.2478	0.2563	0.2043
16	0.2467	0.2453	0.2465	0.1943
17	0.2482	**0.2452**	0.2425	0.1970
18	0.2504	0.2457	0.2394	**0.1913**
19	0.2536	0.2469	**0.2367**	0.2097
20	0.2567	0.2483	0.2368	0.2034

Table 5.2: Test loss - MNIST experiments

Structure	SNN	DDTree-2	DDTree-3	DDTree-4
Model accuracy	94.01 %	92.44 %	93.21 %	93.71 %

Table 5.3: Model accuracy - MNIST experiments

These scores reconfirm the previous results. Apart from what happens with SNN for the reasons mentioned earlier, when we increase the complexity of the DDTree, the accuracy is higher, which is what we should expect.

All of these facts support the usefulness and interest of deciduous decision trees as a new prediction model, aimed at optimizing interpretability and precision at the same time by allowing to select a specific value for one of the traits and maximize the other.

6

Conclusions and future work

The main conclusions that can be extracted from this work are:

- **Machine learning** along with big data are the two most predominant fields in computer science nowadays, with companies investing heavily in their development and showing promising results.
- **Neural networks** are the most famous machine learning model by a large margin, specially because of the precision they are able to obtain, but their development is slowing down because it is difficult to optimize them more than they already are.
- Even if neural networks are really precise, they lack **interpretability**, a trait that might be interesting for some particular cases (medical diagnoses for example) even if obtaining it means getting less precision.
- There is a dichotomy between precision and interpretability that makes it impossible to optimize both of them at the same time.
- It is possible to build a custom model with which we can choose whether to prioritize precision or interpretability and in what degree: the **decidiuous decision tree** is a promising model with a relatively natural implementation and that has achieved good results in terms of accuracy.

In terms of future work, there are two specific strands of work:

- **Deciduous decision tree optimization**. The first aspect of the implementation of the deciduous decision tree that needs to be optimized is clearly the execution time. There are some possible improvements that have been discussed in previous sections, such as the transformation of recursive calculations into matrix operations, so that we can take advantage of the Tensorflow math libraries.

  In addition, new types of tasks and models can be created, with the goal of making the deciduous decision tree behave exactly like any other model in the Tensorflow libraries.

  Finally, other loss functions or optimizers could be designed to improve the performance and the precision of the deciduous decision tree, adapting everything to the nature of its mathematical background.

- **Formal definition and theory on precision and interpretability**. In this work we have not defined formally what “precision” and “interpretability” mean, and the development of a mathematical theoretical background for this pair of concepts could be crucially important for the future of machine learning, specially after neural networks development hits the ceiling.

  The problem is the difficulty at defining what “interpretability” means and at determining whether a model is highly interpretable. Precision can be measured as the opposite of a loss function or the rate of success at classifying, but at the moment we do not even have a candidate for a formal definition of interpretability.

Bibliography

[1] S. Collins, *The Big Data Market: 2018 – 2030 – Opportunities, Challenges, Strategies, Industry Verticals & Forecasts*. Market Reports Center, July 2018. (Link: https://www.globenewswire.com/news-release/2018/07/02/1532157/0/en/Big-Data-investments-will-account-for-over-65-Billion-in-2018-further-expected-to-grow-at-a-CAGR-of-14-over-the-next-three-years.html).

[2] C. Osborne, *Investments in Big Data fuelled by a fear of data-driven competitors*. ZDNet, January 2019. (Link: https://www.zdnet.com/article/fortune-1000-to-urgently-invest-in-big-data-ai-in-2019-in-fear-of-digital-rivals/).

[3] A. Moltzau, *Artificial Intelligence and Recent Billion Dollar Investments 2019*. Medium, August 2019. (Link: https://medium.com/dataseries/artificial-intelligence-and-recent-billion-dollar-investments-2019-759e78b042ad).

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. (Link: http://www.deeplearningbook.org).

[5] T. Mitchell, *Machine learning*. McGraw Hill, 1997.

[6] M. Nielsen, *Neural Networks and Deep Learning*. December 2019. (Link: http://neuralnetworksanddeeplearning.com).

[7] C. Lemaréchal, *Documenta Mathematica*, pp. 251–254. Journal der Deutschen Mathematiker-Vereinigung (DMV), 1996.

[8] DotCSV, *¿Qué es una Red Neuronal? Parte 3.5: Las Matemáticas de Backpropagation | DotCSV*. YouTube, October 2018. (Link: https://youtu.be/M5QHwkkHgAA).

[9] *Why Tensorflow*. Google. (Link: https://www.tensorflow.org/about).

[10] F. Chollet, *Deep learning with Python*. Manning, 2018.

Appendices

A

Implementation: code

A.1. Logic unit: stump

Code A.1: Implementation of the logic unit of a DDTree: the stump

```
1  class Stump(tf.keras.layers.Layer):
2      def __init__(self, s, input_shape):
3          super().__init__()
4
5          # Width constant 's'
6          assert(isinstance(s, float) and s > 0)
7          self.s = tf.constant(s)
8
9          # Dimension of 'b' and input vector 'x'
10         dim_vector = input_shape.as_list()[1]
11
12         # Initialization of trainable weights -uniform distribution
13         initializer = tf.random_uniform_initializer()
14
15         # Trainable weights 'b' and 'a'
16         self.b = self.add_weight("b", shape=[dim_vector], trainable=True,
17                                  initializer=initializer)
18         self.a = self.add_weight("a", trainable=True, initializer=initializer)
19
20     @tf.function
21     def call(self, inputs, **kwargs):
22         """Calculates membership degrees of children nodes
23         :param inputs: input vector
24         :param kwargs: other parameters
25         :return: membership degrees of children nodes
26         """
27         target = tf.tensordot(inputs, self.b, axes=1)
28         target = tf.subtract(target, self.a)
29         target = tf.divide(target, self.s)
30         right_tensor = tf.sigmoid(target)
31         left_tensor = tf.subtract(tf.constant(1.0), right_tensor)
32
33         return left_tensor, right_tensor
```

A.2. Deciduous decision tree

Code A.2: Implementation of the deciduous decision tree: node (abstract)

```
15  class DeciduousDecisionTreeAbstractNode(ABC):
16      def __init__(self, parent, bitchain=None):
17          super().__init__()
18          self.parent = parent
19          if bitchain is None:
20              self.bitchain = NoDependency([])
21          else:
22              self.bitchain = bitchain
23          assert(isinstance(self.bitchain, list))
24
25      @abstractmethod
26      def is_leaf(self):
27          pass
```

Code A.3: Implementation of the deciduous decision tree: decision node

```
30  class DecisionNode(DeciduousDecisionTreeAbstractNode):
31      def __init__(self, parent, bitchain=None):
32          super().__init__(parent, bitchain)
33          self.layer = None
34          self.child_left = None
35          self.child_right = None
36
37      def set_child_left(self, child_left):
38          assert(isinstance(child_left, DeciduousDecisionTreeAbstractNode))
39          self.child_left = child_left
40
41      def set_child_right(self, child_right):
42          assert(isinstance(child_right, DeciduousDecisionTreeAbstractNode))
43          self.child_right = child_right
44
45      def is_leaf(self):
46          return False
```

Code A.4: Implementation of the deciduous decision tree: deciduous node

```
49  class DeciduousNode(DeciduousDecisionTreeAbstractNode):
50      def __init__(self, parent, bitchain):
51          super().__init__(parent, bitchain)
52          self.model = None
53  
54      def build_model(self, input_shape, model, shapes, activator_end,
55                      activator_mid=tf.nn.relu):
56          if isinstance(model, tf.keras.Model):
57              self.model = model
58          elif model == 'NeuralNetwork' or model == 'nn':
59              self._build_neural_network(input_shape, shapes, activator_end,
60                                         activator_mid)
61          else:
62              raise AssertionError
63  
64      def _build_neural_network(self, input_shape, shapes, activator_end,
65                                activator_mid, optimizer='rmsprop', loss='mse'):
66          self.model = tf.keras.Sequential()
67          kernel_initializer = tf.random_uniform_initializer
68          bias_initializer = tf.random_uniform_initializer
69          for i in range(len(shapes)-1):
70              if i == 0 and len(shapes) -1 == 1:
71                  self.model.add(Dense(shapes[i+1],
72                                       input_shape=(shapes[i].astype(np.int),),
73                                       activation=activator_end,
74                                       kernel_initializer=kernel_initializer,
75                                       bias_initializer=bias_initializer))
76              elif i == 0 and len(shapes) -1 > 1:
77                  self.model.add(Dense(shapes[i+1],
78                                       input_shape=(shapes[i].astype(np.int),),
79                                       activation=activator_mid,
80                                       kernel_initializer=kernel_initializer,
81                                       bias_initializer=bias_initializer))
82              elif i == len(shapes) -2:
83                  self.model.add(Dense(shapes[i+1], activation=activator_end,
84                                       kernel_initializer=kernel_initializer,
85                                       bias_initializer=bias_initializer))
86              else:
87                  self.model.add(Dense(shapes[i+1], activation=activator_mid,
88                                       kernel_initializer=kernel_initializer,
89                                       bias_initializer=bias_initializer))
90          self.model.compile(optimizer=optimizer, loss=loss)
91  
92      def is_leaf(self):
93          return True
```

Code A.5: Implementation of the deciduous decision tree: training (1)

```
276         self.reset_weights(self)
277
278         loss_metric = None
279         mean_losses = None
280         if 'loss' in self.ddt_metrics:
281             loss_metric = tf.keras.metrics.Mean()
282             mean_losses = []
283
284         accuracy_metric = None
285         mean_accuracies = None
286         if 'acc' in self.ddt_metrics:
287             accuracy_metric = tf.keras.metrics.Mean()
288             mean_accuracies = []
289
290         time_metric = tf.keras.metrics.Mean()
291
292         x_train_dataset = tf.data.Dataset.from_tensor_slices(x)
293         x_train_dataset = x_train_dataset.batch(batch_size)
294         y_train_dataset = tf.data.Dataset.from_tensor_slices(y)
295         y_train_dataset = y_train_dataset.batch(batch_size)
296
297         n_samples = len(x) // batch_size + 1
298
299         val_losses = None
300         val_accuracies = None
301         x_test = None
302         y_test = None
303         if validation_data is not None:
304             x_test = tf.constant(validation_data[0])
305             y_test = tf.constant(validation_data[1])
306             if 'val_loss' in self.ddt_metrics:
307                 val_losses = []
308             if 'val_acc' in self.ddt_metrics:
309                 val_accuracies = []
310
311         for epoch in range(epochs):
```

Code A.6: Implementation of the deciduous decision tree: training (2)

```
312             print('Start of epoch %d/%d' % (epoch+1, epochs))
313             start = timer()
314
315             if 'loss' in self.ddt_metrics:
316                 loss_metric.reset_states()
317             if 'acc' in self.ddt_metrics:
318                 accuracy_metric.reset_states()
319
320             for (step, x_batch), (_, y_batch) in zip(enumerate(
321                     x_train_dataset), enumerate(y_train_dataset)):
322                 start_batch = timer()
323
324                 with tf.GradientTape() as tape:
325                     pred = self(x_batch, **kwargs)
326                     loss = loss_function(y_batch, pred)
327
328                 grads = tape.gradient(loss, self.trainable_variables)
329                 optimizer.apply_gradients(zip(grads, self.trainable_variables))
330
331                 if 'loss' in self.ddt_metrics:
332                     loss_metric(loss)
333                 if 'acc' in self.ddt_metrics:
334                     accuracy = self.accuracy_fn(pred, y_batch)
335                     accuracy_metric(accuracy)
336
337                 end_batch = timer()
338                 time_metric(end_batch -start_batch)
339
340                 if (step + 1) % 100 == 0:
341                     time = time_metric.result().numpy() / batch_size
342                     print('\tProgress: %d/%d - Mean time per sample: %.5fs' %
343                             (step + 1, n_samples, time))
344                     time_metric.reset_states()
345
346             if validation_data is not None:
347                 val_pred = self(x_test, **kwargs)
```

Code A.7: Implementation of the deciduous decision tree: training (3)

```
348                 if 'val_loss' in self.ddt_metrics:
349                     val_loss = loss_function(y_test, val_pred)
350                     val_losses.append(val_loss.numpy())
351                 if 'val_acc' in self.ddt_metrics:
352                     val_accuracy = self.accuracy_fn(val_pred, y_test)
353                     val_accuracies.append(val_accuracy)
354
355             if 'loss' in self.ddt_metrics:
356                 mean_losses.append(loss_metric.result().numpy())
357             if 'acc' in self.ddt_metrics:
358                 mean_accuracies.append(accuracy_metric.result().numpy())
359
360             end = timer()
361             print('End of epoch %d/%d - Time: %.5fs' % (epoch+1, epochs,
362                                                         end -start))
363
364         history = {}
365         if 'loss' in self.ddt_metrics:
366             history['loss'] = tf.constant(mean_losses)
367         if 'acc' in self.ddt_metrics:
368             history['acc'] = tf.constant(mean_accuracies)
369         if 'val_loss' in self.ddt_metrics:
370             history['val_loss'] = tf.constant(val_losses)
371         if 'val_acc' in self.ddt_metrics:
372             history['val_acc'] = tf.constant(val_accuracies)
373
374         return history
```