Universitat Autònoma de Barcelona

Facultat de Ciències

Departament de Matemàtiques

Artificial Intelligence as a Guide for Mathematical Intuition

Bachelor's Degree Final Project

Andreu Quesada Jaén

Supervised by Dr. Roberto Rubio Núñez

July 2023

Aknowledgments

I would like to thank my supervisor Roberto Rubio Núñez for all the guidance, useful comments and implication during the development of this project. Without him, it would not have been possible.

I would also like to express my gratitude towards my family and friends for the encouragement given, not only during the time this dissertation was written, but for all the previous moments that lead to this point.

Abstract

The aim of this dissertation is to study whether or not artificial intelligence can be a useful tool for theoretical mathematicians as a guide for intuition. We will begin by laying out the required theory, regarding group and game theory. Afterwards, we will explain the mathematical background of neural networks. We will finish with two practical studies that can be seen as an example of the application of machine learning to gain mathematical insight.

Contents

Chapter 1

Introduction

Artificial intelligence and machine learning are current trends. Applications and programs that can be part of the daily routine of many professionals and enhance their capabilities and productivity have started to become popular. A mathematician can integrate these general-public into his or her routine. But, are there other ways that this new technology can been used in more theoretical processes? In this chapter we will explore this possibility.

1.1 Motivation

From the year 2018, Google DeepMind, the subsidiary of Alphabet engaged in developing machine learning, is collaborating with various groups of mathematicians to explore the possibilities of using artificial intelligence as a guide for mathematical intuition. The results of this alliance have resulted in various new developments, [DVea21]. In the project, two different research groups study the following mathematical fields: knot theory and representation theory.

Regarding the former, the hypothesis explored was that there exists a relationship between geometric and algebraic invariants of a knot. A large dataset of geometric invariants for different knots was generated and a machine learning model was trained to predict the signature of the knot, an algebraic invariant, from the geometric invariants. The model was able to make accurate predictions, furthering the validity of the hypothesis. However, this by itself is not a conclusive result, since with a big enough neural network any function can be approximated, as we will later see. After there were determined the most significant geometric invariants for the model to make the prediction and a couple of conjectures were made, a theorem that links both types of invariants was proven. This has claimed to be the first successful connection between algebraic and geometric knot theory.

When it comes to representation theory, a similar scheme was followed. A machine learning model was trained to predict the so-called Kazhdan-Lusztig polynomial from the Bruhat interval. Then, analysing the model that predicted the polynomial enabled to gain insight about the most essential parts of the problem, which allowed to prove a theorem related to what is known as the combinatorial invariance conjecture for symmetric groups.

Overlooking the particular mathematical details of the cases studied, there are some lessons to be learned. First of all, a supervised neural network, which is a type of machine learning model, was used in both problems. In addition, the central part of the work was not training an algorithm to predict the considered values, but rather analyzing the model to understand which were the essential inputs. This is the process that led to stating the conjectures and then mathematically proving the results, rather than building a model that works properly in millions of examples. The role of the algorithm was to guide mathematicians and providing insight that wasn't previously available. From all this, it is reasonable to assume that if artificial intelligence techniques are to be applied, a good starting point would be to consider a problem where it is possible to build a proper dataset suitable for a supervised neural network as well as looking into how the model will be interpreted.

1.2 Objectives

In this project we are interested in studying how can artificial intelligence help theoretical mathematics based on the work we have discussed, rather than its use in more applied fields. That means that the goal is not only obtaining effective models, but also being able to interpret the results and relationships found by the programs. The way we have decided to do so is by considering a couple of particular problems where we found a strategy to apply machine learning techniques with the considerations mentioned above. Both problems involve finite groups and their properties. In the first one, we will investigate whether or not a neural network can differentiate nilpotent and non-nilpotent groups from its Cayley table. In the second, we will explore if a neural network is able to find any interesting relationship among various group properties. In both cases we will try to analyze the trained model and interpret the results.

The structure of this dissertation is as follows. We first lay out the mathematical concepts regarding group and game theory required to understand the problems and their analysis. Secondly, we will explain the basic concepts of machine learning. In particular, we are going to describe supervised neural networks and briefly review the training process as well as two ways of interpreting the models. Finally, we will present the problems studied, how machine learning was applied to those and what are the results obtained.

Chapter 2

Theoretical Background

In this chapter we are going to explain the mathematical theory required to develop the project and analyze the models. It is not meant to be a thorough mathematical scrutiny with the corresponding proofs, but rather a compendium of the essential concepts.

2.1 Group Theory

Since both practical examples involve groups, we will give a quick review of the basic definitions and important results, as well as some of the properties that make up the databases.

2.1.1 Elemental Notions

Let us fix the notation by recalling some basic definitions.

Definition 1. Let G be a set and $\cdot : G \times G \to G$ a binary operation. Then, the pair $(G, \cdot)$ is a **group** if the following properties are satisfied:

1. For all $a, b, c \in G$, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.
2. There exists an element $e \in G$ such that for all $a \in G$, $a \cdot e = e \cdot a = a$.
3. For each $a \in G$ exists an element $b \in G$ such that $a \cdot b = b \cdot a = e$. This element is denoted by a^{-1}.

Furthermore, if $(G, \cdot)$ is a group, the **order of a group** is the number of elements in G, it will be denoted by $|G|$. We say that G is a **commutative group** if $a \cdot b = b \cdot a$ for all $a, b \in G$. We will usually denote the operation by concatenation.

Definition 2. Let $(G, \cdot)$ be a group and $a \in G$, The **order** of a is the smallest $n \in \mathbb{N}$ such that $a^n = e$.

Definition 3. Let $(G, \cdot)$ be a group and $H \subseteq G$ a subset of G. Then H is a **subgroup** of G if the pair $(H, \cdot)$ is a group. That H is a subgroup of G is denoted by $H \leq G$.

The following type of subgroups are particularly relevant.

Definition 4. Let $(G, \cdot)$ be a group and $N \leq G$ a subgroup of G. Then N is a **normal subgroup** if $gng^{-1} \in N$ for all $n \in N$ and $g \in G$. That N is a normal subgroup of G is denoted by $N \trianglelefteq G$.

Let G be a group. Let $H \leq G$ be a subgroup of G. For $g \in G$ the **left coset** is $gH = \{gh \mid h \in H\}$. This definition and normality define the quotient construction:

Proposition 5. *Let G be a group and $N \leq G$ a normal subgroup of G. Then the set $G/N = \{gN \mid g \in G\}$ with the operation $(aN)(bN) = (ab)N$ is a group.*

We can describe a group by different means, and one that will be of interest for finite groups is the following:

Definition 6. Let $G = \{a_1, \ldots, a_n\}$ be a finite group. The **Cayley table** of group G is the table such that the element of the $i-$th row and $j-$th column is $a_i \cdot a_j$.

Another way of describing a group is the following:

Definition 7. A **presentation of a group** G is a expression of the type $G = \langle S \mid R \rangle$, where S is the set of generators of G and R is the set of relations among the generators that define G.

Example 8. The **quaternion group** is defined as $Q_8 = \langle a, b \mid a^4 = e, a^2 = b^2, ba = a^{-1}b \rangle$. The **dihedral group** is $D_n = \langle r, s \mid r^n = s^2 = (sr)^2 = e \rangle$.

Given a group that is not necessarily commutative, we are interested in seeing how far from this property is.

Definition 9. Let G be a group and $a, b \in G$. The **commutator** of a and b is defined as $[a, b] = aba^{-1}b^{-1}$.

Definition 10. Let G be a group. The **commutator subgroup** of G is the set defined by $[G, G] = \{[a_1, b_1] \cdots [a_n, b_n] \mid a_1, \ldots, a_n, b_1, \ldots, b_n \in G\}$. Let $H \leq G$ be a subgroup of G. Then $[G, H] = \{[a_1, b_1] \cdots [a_n, b_n] \mid a_1, \ldots, a_n \in G, b_1, \ldots, b_n \in H\}$.

The following result is of interest:

Proposition 11. *Let G be a group. Then $[G, G]$ is a normal subgroup of G and $G/[G, G]$ is commutative. Furthermore, if $N \trianglelefteq G$ is a normal subgroup of G, then G/N is commutative if and only if $[G, G] \subseteq N$.*

The next relationship will be used in our practical examples.

Definition 12. Let G be a group and $\sim$ the equivalence relationship defined by $a \sim b$ if and only if there exists $g \in G$ such that $gag^{-1} = b$. Then a and b are said to be **conjugate**. The equivalence classes of the relationship are named **conjugacy classes**. The number of equivalence classes in a group will be denoted by $\kappa(G)$.

Proposition 13. *Let G and H be groups. Then, the set $G \times H = \{(g, h) \mid g \in G, h \in H\}$ with the binary operation $(g_1, h_1) \cdot (g_2, h_2) = (g_1 \cdot g_2, h_1 \cdot h_2)$ is a group.*

Proposition 14. *Let G and H be groups. Then $(g_1, h_1), (g_2, h_2) \in G \times H$ are conjugate if and only if g_1 and g_2 are conjugate in G and h_1 and h_2 are conjugate in H. Moreover, $\kappa(G \times H) = \kappa(G) \cdot \kappa(H)$.*

2.1.2 Nilpotent Groups

A recurring property in the empirical study will be nilpotency. Let us consider the following notions:

Definition 15. Let G be a group. A **subgroup series** of G is a sequence $A_0 \leq A_1 \leq \cdots \leq A_n$ such that $A_0, A_1, \ldots, A_n \leq G$ are subgroups of G, $A_0 = \{e\}$ and $A_n = G$. If each subgroup is a normal subgroup of the following one, the sequence is a **normal series** which is denoted by $A_0 \trianglelefteq A_1 \trianglelefteq \cdots \trianglelefteq A_n$.

Definition 16. Let G be a group and $A_0 \trianglelefteq A_1 \trianglelefteq \cdots \trianglelefteq A_n$ a normal series in G. Then the sequence is a **central series** if $[G, A_{i+1}] \leq A_i$ for $i \in \{1, \ldots, n-1\}$.

With this, we have all the ingredients to define nilpotent groups.

Definition 17. A group is **nilpotent** if it has a central series of finite length. The length of the shortest possible central series of a nilpotent group is the **nilpotency class** of the group.

Example 18. Commutative groups are nilpotent.

The main intuition, which we are not going to discuss, is that nilpotent groups are a generalization of commutative groups. There is a special case where it's easy to see if a group is nilpotent given its order.

Definition 19. Let G be a group. If G has p^k elements, where $p \in \mathbb{N}$ is a prime and $k \in \mathbb{N}$, then G is a $p-$**group**.

Theorem 20. *Let $p \in \mathbb{N}$ be a prime and let G be a finite $p-$group. Then, G is nilpotent.*

We can find a proof of this result in [CMZ18, Thm. 2.3]. The property of being nilpotent also applies to subgroups, and in what follows we will see two ways to define nilpotent subgroups.

Definition 21. Let G be a group. Let $H \leq G$ be a subgroup of G such that $H \subsetneq G$. Let $K \leq G$ be a subgroup of G. If $H \leq K \leq G$ implies that $H = K$ or $K = G$, then H is a **maximal subgroup** of G.

Definition 22. Let G be a group. The **Frattini subgroup** of G is defined as the intersection of all maximal subgroups of G. It is denoted by $\phi(G)$.

Theorem 23. *Let G be a finite group. Then, the Frattini subgroup $\phi(G)$ is a nilpotent group.*

This theorem is proven in [CMZ18, Cor. 7.8]. Let's consider one last definition, also related to nilpotency.

Definition 24. Let G be a group. The **Fitting subgroup** of G is defined as the subgroup generated by all the normal nilpotent subgroups of G. It is denoted by $F(G)$.

Theorem 25. *Let G be a finite group. Then, the Fitting subgroup $F(G)$ is the largest normal nilpotent subgroup.*

A proof of this last theorem can be found in [CMZ18, Lem. 7.18]. The references consulted for this section have been [Coh12] and [CMZ18].

2.2 Game Theory

There is an unexpected connection between game theory and interpretation of machine learning models that will be useful in this project. To understand it, we'll first comment about its theoretical justification.

2.2.1 Elemental Notions

Game theory is a formalization of rational interactions among agents in a specific context. The agents will be referred to as **players** and the context as **game**. It has had a big influence in many areas of studies where logical decisions have to be made, since its modern characterization last century by John von Neumann.

There are several ways to describe a game. If we want it to be fully specified, the players, the available information, the possible actions and the reward or **payoff** have to be defined. There are **non-cooperative games**, in which players do not necessarily cooperate, but we will focus on **cooperative games**. In this kind of games, players share a specific goal and compete with each other, but can also form groups and make contracts as they will. We will be interested in studying which groups are formed and what are their payoffs, without focusing on the information available and the possible actions.

To formalize this idea, let's introduce the following notions:

Definition 26. Let $\mathcal{S}$ be a set of n players. Then, a **coalition** is a subset of players $C \subseteq \mathcal{S}$. The coalition $\mathcal{S}$ is known as the **grand coalition**.

Definition 27. Let $\mathcal{S}$ be a set of n players. A **characteristic function** of an $n-$players game is a function $v : \mathcal{P}(\mathcal{S}) \to \mathbb{R}$ such that:

1. $v(\emptyset) = 0$
2. If $C, D \in \mathcal{P}(S)$ are disjoint, $v(C) + v(D) \leq v(C \cup D)$.

The characteristic function is interpreted as a way of assigning the value or payoff of each coalition. Thus, it is used to define a cooperative game:

Definition 28. Let $\mathcal{S}$ be the set of players of an $n-$players game and let $v : \mathcal{P}(\mathcal{S}) \to \mathbb{R}$ be its characteristic function. Then $(\mathcal{S}, v)$ is the **coalitional form** of the game.

There could be games were players are not inclined to form coalitions. This is why we make the following distinction:

Definition 29. Let $(\mathcal{S}, v)$ be the coalitional form of an $n-$players game. The game is **inessential** if $\sum_{a \in \mathcal{S}} v(\{a\}) = v(\mathcal{S})$. The game is **essential** if $\sum_{a \in \mathcal{S}} v(\{a\}) < v(\mathcal{S})$.

This means that essential games will be of greater interest when the aim is to study coalitions.

2.2.2 Shapley Values

The characteristic function attributes a value to every coalition. But we may be interested in defining which was the contribution of each player to the coalition. For this, we have

to split the payoff, and this should be done in a fair manner.

Definition 30. Let $(\mathcal{S}, v)$ be the coalitional form of an $n-$players game. A **value function** of the game is a function $\phi(v) : \mathcal{S} \to \mathbb{R}$, whose images $\phi(v)(a)$ will be denoted by $\phi_a(v)$ for $a \in \mathcal{S}$, such that:

1. $\sum_{a \in \mathcal{S}} \phi_a(v) = v(\mathcal{S})$
2. Let $a, b \in \mathcal{S}$ such that $v(C \cup \{a\}) = v(C \cup \{b\})$ for every $C \subseteq \mathcal{S} \setminus \{a, b\}$. Then, $\phi_a(v) = \phi_a(v)$.
3. Let $a \in \mathcal{S}$ such that $v(C) = v(C \cup \{a\})$ for every $C \subseteq \mathcal{S} \setminus \{a\}$. Then, $\phi_a(v) = 0$.
4. Let $u : \mathcal{P}(\mathcal{S}) \to \mathbb{R}$ be another characteristic function. Then $\phi(u + v) = \phi(u) + \phi(v)$.

We will consider the properties of a value function as the definition of **fairness**: the total payoff is split; if two players contribute the same to every coalition, they receive the same reward; if a player doesn't contribute gets no value; and there is no difference if two uncorrelated games are played simultaneously or consecutively.

Theorem 31 (Shapley)**.** *Let (S, v) be the coalitional form of an $n-$players game. Then, there exists a unique value function of the game $\phi(v) : \mathcal{S} \to \mathbb{R}^n$ defined, for $a \in \mathcal{S}$, by:*

$$\phi_a(v) = \sum_{C \subseteq \mathcal{S} \setminus \{a\}} \frac{|C|!\,(n - |C| - 1)!}{n!} \cdot (v(C \cup \{a\}) - v(C)) \,.$$

The values $\phi_a(v)$ for $a \in \mathcal{S}$ are know as **Shapley values**. The idea is weighing the marginal contribution of the player to a coalition with the probability of this coalition being formed, and adding over all possible coalitions that don't include the player.

Notice that there is an important difference between $\phi_a(v)$ and $v(\{a\})$. The former assigns a fair division of the payoff to player $a \in \mathcal{S}$ when the game is played by the grand coalition $\mathcal{S}$. The latter is player's a payoff when plays the game by itself.

The references consulted for this section have been [Sha52] and [Fer00].

Chapter 3

Neural Networks

In this chapter we will provide the most basic definitions and explanations to understand what is supervised machine learning, in order to later develop and examine the cases we propose. In particular, we will focus on neural networks, explaining how they work as well as describing some of the characteristics that affect their performance.

3.1 Elemental Notions

Whether it is classifying images of cats and dogs, predicting algebraic invariants, or even imitating human written language, we can summarize the role performed by supervised machine learning techniques as finding an unknown function, in the broadest sense of the word function, and improving certain predictions or behaviors using some known dataset.

Schematically, the procedure is as follows: to solve the problem at hand, we want to construct a specific function. Using a set of sample data and applying supervised machine learning algorithms, we will obtain an approximation of this function. The goal is that this function acts on new data, up to a certain degree of accuracy that depends on the particular case, in the expected way. The peculiarity and usefulness of the method lies in that we don't need to program an algorithm to solve each specific case, rather the parameters of the model adapt autonomously based on the dataset that includes the outcomes of interest. To achieve this, the model must be trained, a procedure that involves minimizing an error function over the sample data. We'll break this down.

Let's suppose we want to approximate a function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ from a set of sample data $D = \{(x^{(j)}, y^{(j)})\}_{j=1}^{M}$, with $x^{(j)} \in \mathbb{R}^n$ and $y^{(j)} = f(x^{(j)}) \in \mathbb{R}^m$, using supervised machine learning techniques. The vectors $x^{(j)}$ will be named **input values** and the $y^{(j)}$ will be the **output values**. What we want is to be able to correctly predict output values corresponding to input values that have not been used to train the model.

A **neural network** is a type of machine learning algorithm made up of a set of elements and parameters with a determined structure that acts on the input values. Next, we will describe the different elements that make up a neural network and how are they interrelated: this constitutes the **model** to be trained.

A representation of a very simple neural network is the one we find in Figure 3.1. As we can see, we have an **input layer** where we input the data $x^{(j)}$ into the model. We also have an **output layer**, where the model returns a result that, as we'll see, will allow us to gain information about the function f. The nodes in these two layers do not perform any operations, they simply serve as the entry point for the data and the exit point for the results.

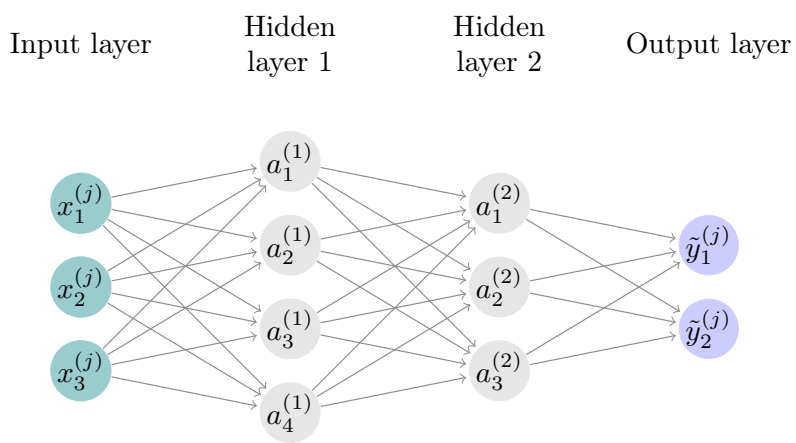


Figure 3.1: A simple neural network.

The different nodes in the **hidden layers** are called **neurons** or **perceptrons**. The number of hidden layers is called the **depth** of the neural network. The number of nodes in each hidden layer is called the **width** of the layer. Thus, in the previous example, we have a neural network with depth 2 and widths 4 and 3, respectively. We will denote the j-th neuron in layer l as $a_j^{(l)}$.

We can think of neurons as the place where operations are performed with the data from the previous layer. Each perceptron performs a linear operation followed by a non-linear operation. The linear operation consists of a linear combination of the data from the previous layer, adding a bias term. The non-linear operation is usually called **activation function** and we will denote it by σ. Examples of the usually used activations functions are $\sigma(x) = \max(0, x)$, $\sigma(x) = \tanh x$ and $\sigma(x) = (1 + e^{-x})^{-1}$.

For example, in the case of neuron $a_1^{(1)}$, the **activation** is $z_1^{(1)} = \sum_{k=1}^{3} w_{1,k}^{(1)} x_k^{(j)} + b_1^{(1)}$, and then $a_1^{(1)} = \sigma(z_1^{(1)})$. In particular, $w_{j,k}^{(l)} \in \mathbb{R}$ is called the **weight** that connects $a_j^{(l)}$ with $a_k^{(l-1)}$, considering that when $l = 0$, we refer to the input values. With this notation, we can write the operations of each layer in matrix form. For example, for the first hidden layer, and understanding that the composition with σ acts on each entry of the matrix,

$$\begin{pmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \\ a_4^{(1)} \end{pmatrix} = \sigma \left(\begin{pmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} & w_{1,3}^{(1)} \\ w_{2,1}^{(1)} & w_{2,2}^{(1)} & w_{2,3}^{(1)} \\ w_{3,1}^{(1)} & w_{3,2}^{(1)} & w_{3,3}^{(1)} \\ w_{4,1}^{(1)} & w_{4,2}^{(1)} & w_{4,3}^{(1)} \end{pmatrix} \begin{pmatrix} x_1^{(j)} \\ x_2^{(j)} \\ x_3^{(j)} \end{pmatrix} + \begin{pmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \\ b_4^{(1)} \end{pmatrix} \right) = \sigma \begin{pmatrix} z_1^{(1)} \\ z_2^{(1)} \\ z_3^{(1)} \\ z_4^{(1)} \end{pmatrix}$$

More generally, if the depth of a network is N and the l-th hidden layer has width n_l for $l \in 1, \ldots, N$, considering that for $l = 0$ we have the input layer with width $n_0 = n$ and

for $l = N+1$ we have the output layer with width $n_{N+1} = m$, it follows that:

$$a^{(l)} = \begin{pmatrix} a_1^{(l)} \\ \vdots \\ a_{n_l}^{(l)} \end{pmatrix} = \sigma \left(\begin{pmatrix} w_{1,1}^{(l)} & \cdots & w_{1,n_{l-1}}^{(l)} \\ \vdots & \ddots & \vdots \\ w_{n_l,1}^{(l)} & \cdots & w_{n_l,n_{l-1}}^{(l)} \end{pmatrix} \begin{pmatrix} a_1^{(l-1)} \\ \vdots \\ a_{n_{l-1}}^{(l-1)} \end{pmatrix} + \begin{pmatrix} b_1^{(l)} \\ \vdots \\ b_{n_l}^{(l)} \end{pmatrix} \right) = \sigma \begin{pmatrix} z_1^{(l)} \\ \vdots \\ z_{n_l}^{(l)} \end{pmatrix} = \sigma(z^{(l)})$$

Taking all into account, for the hidden layer l we will have $p_l = n_l \cdot n_{l-1} + n_l$ parameters. We will denote by p the number of all parameters in the neural network. That means that in total we have $p = \sum_{l=1}^{N+1} n_l \cdot (1 + n_{l-1})$ parameters.

Each set of $N+1$ matrices of weights and biases defines a function $\tilde{f} : \mathbb{R}^n \to \mathbb{R}^m$ that represents the neural network. The class of all such functions will be the **class of neural networks** defined by its dimensions, denoted by $\mathcal{N}_{n,m,N}^{n_1,\ldots,n_N,\sigma}$. When we are not interested in specifying the widths, we will write $\mathcal{N}_{n,m,N}^{\sigma}$. This type of neural networks without any additional restriction are also known as **fully connected neural networks**.

The **parameters** of the network (weights $w_{j,k}^{(l)}$ and biases $b_j^{(l)}$) are the model variables that are modified during training. The **hyperparameters** are all the other variables of the neural network that define it but are not learned during the training process. For example, **structure** of the neural network (depth, widths) or the activation function.

Once the problem is formulated, the sample dataset D is collected, and the neural network is defined, it is necessary to train the model. We will divide the dataset D into two disjoint subsets, one for training, D_t of size M_t, and the other for validating how the network behaves against data it has never seen before, D_v of size M_v. This division is made following different criteria, depending on the problem we are facing. As a rule of thumb, more data is usually allocated for training than for validation.

Introducing a training data vector $x^{(j)}$ to the input layer, we obtain a vector $\tilde{f}(x^{(j)}) = \tilde{y}^{(j)} \in \mathbb{R}^m$, this is known as **forward propagation**. We want $\tilde{y}^{(j)} = a_j^{(L+1)}$ to be as similar as possible to the output value $f(x^{(j)}) = y^{(j)}$. This is what we need to achieve for the elements of D_t through training, and this is what we mean when we talk about optimizing the parameters. The goal is that the good results are transferred to the vectors of D_v and that the model also functions for inputs outside the database, which is know as **generalization**. This is, of course, not guaranteed.

In conclusion, we need to measure how well our model performs on the data in D_t, and we will do this according to the information provided by a **loss function** $L(\tilde{f}) : \mathbb{R}^p \longrightarrow \mathbb{R}$, that has as variables the parameters that define $\tilde{f}$. Then, we aim to minimize this function, that represents the error, so we must adjust the parameters of the network accordingly. For this purpose, gradient descent and backpropagation are used since analytical methods are not fruitful, due to the high number of parameters.

The references for this section have been Weeks 1 and 2 of [GGW22], [San18] as well as Sections 5.1, 5.2, 5.7, 6.1 of [GBC16]

3.2 Training

We will explain how to enhance the performance of $\tilde{f}$ through the information given by $L(\tilde{f})$. Note that we are not optimizing the parameters that define $\tilde{f}$ directly, we are doing so through $L(\tilde{f})$, which depends on the parameters: if we minimize the loss, then we will have a model with better performance. We can talk about **training loss** when the loss is computed using the inputs corresponding to the training dataset, or **validation loss** when the validation dataset is used.

The loss function has to be appropriate for the particular case, and the standard choices are the following. For regression problems, that is, when we try to predict a real number (for example, the price of a second-hand car given its characteristics), **mean squared error** is widely used. In this case, usually $m = 1$ and the loss function is:

$$L(\tilde{f}) = \frac{1}{M_x} \sum_{x^{(j)} \in D_x} \sum_{k=1}^{m} \left(f_k\left(x^{(j)}\right) - \tilde{f}_k\left(x^{(j)}\right) \right)^2 .$$

In contrast, for classification problems where a probability distribution prediction must be inferred, **cross entropy** is the general adoption, which is based on the principle of maximum likelihood:

$$L(\tilde{f}) = -\frac{1}{M_x} \sum_{x^{(j)} \in D_x} \sum_{k=1}^{m} f_k\left(x^{(j)}\right) \cdot \log\left(\tilde{f}_k\left(x^{(j)}\right) \right) .$$

The idea is that answers that are confident and wrong contribute more negatively to the loss function. Here, M_x refers to the number of elements in D_x, which can be D_t or D_v. Also, the subindex k indicates the $k-$th component of the vector, and all the $x^{(j)}$ are training inputs. Thus, $\tilde{f}_k(x^{(j)})$ is the probability or score the model assigns to the input belonging in class k. Note that in this last case, for binary classification problems, $f_k(x^{(j)}) \in \{0, 1\}$.

3.2.1 Optimization

Theoretically, convex optimization converges regardless of the starting parameters. Neural networks' non-linearity, however, implies that loss functions as the ones defined above are generally non-convex. This means that the algorithms are not guaranteed to converge. In practice, we want to reduce the error to an amount that makes the model useful. Gradient descent and its variations are the most extended methods for their good results in training machine learning algorithms.

Gradient Descent

The mathematical idea behind **gradient descent** is simple. Assuming that the loss function is differentiable, let $\nabla L(\tilde{f}^{(0)}) \in \mathbb{R}^p$ be the gradient of $L(\tilde{f})$ evaluated at the parameters of $\tilde{f}^{(0)}$. We know it indicates the direction of fastest increase of the function at that point. This means that $L(\tilde{f}^{(1)}) \leq L(\tilde{f}^{(0)})$ if $\tilde{f}^{(1)} = \tilde{f}^{(0)} - \alpha \cdot \nabla L(\tilde{f}^{(0)})$, for small enough $\alpha > 0$. So this iterative method consists in making an initial selection of the

parameters $\tilde{f}^{(0)}$ and applying the iteration $\tilde{f}^{(i+1)} = \tilde{f}^{(i)} - \alpha_i \cdot \nabla L(\tilde{f}^{(i)})$ to update the parameters. This $\alpha^{(i)} > 0$ is known as **learning rate**.

The success of the method depends strongly on the value given to the learning rate. It can remain constant, but a simple way of updating is the following: if $L(\tilde{f}^{(i+1)}) > L(\tilde{f}^{(i)})$, this means that the rate $\alpha^{(i)}$ was too large. That step is repeated with the learning size $\alpha^{(i)}$ decreased until $L(\tilde{f}^{(i+1)}) \leq L(\tilde{f}^{(i)})$. In contrast, $L(\tilde{f}^{(i+1)}) < L(\tilde{f}^{(i)})$ might mean the rate could have been bigger, so we will define $\alpha^{(i+1)} > \alpha^{(i)}$.

We also need a criterion to know when to stop iterating. This can be fixing a number of maximum iterations or stopping when the loss is smaller than a fixed value. Another way is to stop when the validation loss stops decreasing for some iterations, even if the training loss still decreases, which prevents the model from memorizing the training dataset, incident known as **overfitting**. Since the aim when training an algorithm is not to find a local minimum, but rather a working model, this last criteria is widely used.

Until now, we have been evaluating the loss function at all the points in D_t, and as we know, a large dataset is necessary for the method to generalize. This means that the more the training values, the more time it takes to compute the loss function and perform each gradient descent step, which is counterproductive. A solution to this problem is **stochastic gradient descent**. We fix the **batch size** $M_b < M_t$. Then, at each step we consider a random sample or **batch** B, consisting of M_b elements of the training set D_t. The new loss function will be, in the case of cross entropy:

$$L(\tilde{f}) = -\frac{1}{M_b} \sum_{x^{(j)} \in B} \sum_{k=1}^{m} f_k\left(x^{(j)}\right) \cdot \log\left(\tilde{f}_k\left(x^{(j)}\right)\right).$$

The corresponding iteration will be $\tilde{f}^{(i+1)} = \tilde{f}^{(i)} - \alpha^{(i)} \cdot \nabla L(\tilde{f}^{(i)})$. Usually, M_b is pretty small when compared to M_t. In this approach, it is required that the learning rate decreases with the iterations. This is because the gradient is approximate and it does not tend to zero at local minimum as quickly as the exact does.

We will define an **epoch** as the number of times the training process will use all the data. This means that in every epoch there will be the closest integer to M/M_b updates of the loss function and the parameters, since the data is not used repeatedly in the same epoch.

Even though following an estimation of the gradient may sound as a pretty rough approximation, it has real practical advantages. First of all, the gradient will be updated more frequently. This means that even though its asymptotic convergence is slower, it will make more progress in the initial steps. More importantly, since M_b is fixed, we can increase the training data without increasing the computation required. This might result in a better-generalizing algorithm.

Another tweak to the gradient descent is known as **momentum**. It is used to increase the velocity of convergence in some cases, and uses the information of previous iterations to do so. The iterations with momentum will be $\tilde{f}^{(i+1)} = \tilde{f}^{(i)} - \alpha^{(i)} \cdot \nabla L(\tilde{f}^{(i)}) + \beta \cdot \Delta\tilde{f}^{(i)}$, where $\Delta\tilde{f}^{(i)} = \tilde{f}^{(i)} - \tilde{f}^{(i-1)}$ and $\beta \in [0, 1)$, even though there are other variations.

The references of this section have been Week 3 of [GGW22], Sections 7.1 of [DFO20], as well as Sections 4.3, 5.9, 6.2, 8.1 and 8.3 of [GBC16].

Backpropagation

The loss function will be computed with the results obtained through forward propagation, and to evaluate its gradient **backpropagation** will be handy. This algorithm is an automatic differentiation method that benefits from the chain rule and intermediate calculations to improve efficiency. As a result, computing the gradient is of a similar complexity as evaluating the function, which is usually not the case with symbolic differentiation.

We want to find

$$\nabla L(\tilde{f}) = \left(\frac{\partial L(\tilde{f})}{\partial w_{1,1}^{(1)}}, \frac{\partial L(\tilde{f})}{\partial b_1^{(1)}}, \dots, \frac{\partial L(\tilde{f})}{\partial w_{m,n_N}^{(N+1)}}, \frac{\partial L(\tilde{f})}{\partial b_m^{(N+1)}} \right).$$

To do so, we will define the following quantity for $j \in \{1, \dots, n_l\}$ and $l \in \{1, \dots, N+1\}$, recalling that $z_j^{(l)} = \sum_{s=1}^{n_{l-1}} w_{js}^{(l)} a_s^{(l-1)} + b_j^{(l)}$:

$$\delta_j^{(l)} = \frac{\partial L(\tilde{f})}{\partial z_j^{(l)}}$$

For the last layer, considering that $\tilde{y}_j = a_j^{(N+1)}$ and using the chain rule, we have that:

$$\delta_j^{(N+1)} = \frac{\partial L(\tilde{f})}{\partial z_j^{(N+1)}} = \sum_{s=1}^{m} \frac{\partial L(\tilde{f})}{\partial a_s^{(N+1)}} \frac{\partial a_s^{(N+1)}}{\partial z_j^{(N+1)}} = \frac{\partial L(\tilde{f})}{\partial a_j^{(N+1)}} \frac{\partial a_j^{(N+1)}}{\partial z_j^{(N+1)}} = \frac{\partial L(\tilde{f})}{\partial \tilde{y}_j} \cdot \sigma'(z_j^{(N+1)})$$

since $a_j^{(N+1)} = \sigma(z_j^{(N+1)})$ and $z_j^{(N+1)} = \sum_{s=1}^{n_N} w_{js}^{(N+1)} a_s^{(N)} + b_j^{(N+1)}$.

Now, let us find $\delta_j^{(l)}$ for $l \in \{1, \dots, N\}$.

$$\delta_j^{(l)} = \frac{\partial L(\tilde{f})}{\partial z_j^{(l)}} = \sum_{s=1}^{n_{l+1}} \frac{\partial L(\tilde{f})}{\partial z_s^{(l+1)}} \frac{\partial z_s^{(l+1)}}{\partial z_j^{(l)}} = \sum_{s=1}^{n_{l+1}} \delta_s^{(l+1)} \cdot \frac{\partial z_s^{(l+1)}}{\partial z_j^{(l)}} = \sum_{s=1}^{n_{l+1}} \delta_s^{(l+1)} \cdot w_{sj}^{(l+1)} \cdot \sigma(z_j^{(l)})$$

With this we can compute the desired derivatives easily. First:

$$\frac{\partial L(\tilde{f})}{\partial w_{jk}^{(l)}} = \sum_{s=1}^{n_l} \frac{\partial L(\tilde{f})}{\partial z_s^{(l)}} \frac{\partial z_s^{(l)}}{\partial w_{jk}^{(l)}} = \frac{\partial L(\tilde{f})}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} \cdot a_k^{(l-1)}$$

Finally we have:

$$\frac{\partial L(\tilde{f})}{b_j^{(l)}} = \sum_{s=1}^{n_l} \frac{\partial L(\tilde{f})}{\partial z_s^{(l)}} \frac{\partial z_s^{(l)}}{b_j^{(l)}} = \frac{\partial L(\tilde{f})}{\partial z_j^{(l)}} = \delta_j^{(l)}.$$

With all this, we can better understand the name backpropagation, since we will be obtaining the gradient with a recursive method, starting from the last layer. Moreover, notice that many factors of the derivatives, such as the activations and weights, are already known or will be used iteratively, and the others are easy to compute, such as the derivative of σ.

The references for this section have been Section 5.6 of [DFO20], Section 6.5 of [GBC16] as well as Chapter 2 in [Nie15].

3.2.2 Initialization

The loss function depends on the p parameters of the network, and to optimize them through the iterations of the method they must be first initialized. Due to the non-convex essence of the problem, it is important to choose appropriate values so that the convergence is possible and effective. It has been found that the initial configuration has also consequences in the model's ability to generalize.

If the initial weights and values are all set to the same value, then all parameters will evolve symmetrically. This is because all neurons will have the same influence on the loss function, as we can see from the equations derived before. Since we want to explore the performance of as many different functions as we can, it then makes sense to initialize the parameters randomly. If the values are too small or too large, the iterative nature of the method will result in divergence due to vanishing or exploding gradients.

To avoid all this, the criteria usually considered is to choose the parameters so that the biases are all the same, the weights are all different, the mean of the initial neurons $\{a_j^{(l)} \mid l \in \{1, \dots, N\} \text{ and } j \in \{1, \dots, n_l\}\}$ is zero and the variance of $\{a_1^{(l)}, \dots, a_{n_l}^{(l)}\}$ is the same for every $l \in \{1, \dots, N\}$. For $\sigma(x) = \max(0, x)$, this can be done choosing the weights of each layer from a normal distribution of mean 0 and variance $\frac{2}{n_{l-1}}$. Another method is choosing the wights from a uniform distribution in the interval $\left(-\frac{1}{\sqrt{n_{l-1}}}, \frac{1}{\sqrt{n_{l-1}}}\right)$.

The references for this subsection have been Section 8.4 of [GBC16] and [HZRS15].

3.3 Justification of the Method

Even though the quality of the final result and accuracy of the neural network depends on the particular problem and on the data with which the model has been trained, we will try to provide some general mathematical justification for the method.

First of all, let's consider some notions that will lead us to the classical universal approximation theorem for neural networks.

Definition 32. Let X and Y be topological spaces. Then $\mathcal{C}(X, Y)$ denotes the set of all continuous functions from X to Y.

Definition 33. Let $N \subseteq \mathcal{C}(\mathbb{R}^n, \mathbb{R}^m)$. Then N is **dense** in $\mathcal{C}(\mathbb{R}^n, \mathbb{R}^m)$ if for every function $f \in \mathcal{C}(\mathbb{R}^n, \mathbb{R}^m)$, every compact set $K \subseteq \mathbb{R}^n$ and every $\varepsilon > 0$, there exists $f \in N$ such that $\max_{x \in K} ||f(x) - g(x)|| < \varepsilon$.

With the notation introduced previously, let's denote now $\mathcal{N}_{n,m,1}^{\infty,\sigma}$ as the class of neural networks of n elements in the input layer, m elements in the output layer and a single hidden layer of arbitrary widht. Then we have the following result by Cybenko, Hornik and Pinkus [Pin99, Thm. 3.1]:

Theorem 34 (Universal Approximation I)**.** *Let $\sigma \in \mathcal{C}(\mathbb{R}, \mathbb{R})$. Let $K \subseteq \mathbb{R}^n$ be a compact set. Then $\mathcal{N}_{n,m,1}^{\infty,\sigma}$ is dense in $\mathcal{C}(K, \mathbb{R}^m)$ if and only if σ is non-polynomial.*

In other words, if we have a neural network of one hidden layer of arbitrary width, we will be able to approximate, theoretically, every continuous function in $\mathcal{C}(\mathbb{R}^n, \mathbb{R}^m)$.

We can define $\mathcal{N}_{n,m,\infty}^{k,\sigma}$ as the class of neural networks of n elements in the input layer, m elements in the output layer, an arbitrary number of hidden layers each of width k and σ as activation function. Now, we have the following result by Kidger and Lyons [KL20, Thm. 3.2]:

Theorem 35 (Universal Approximation II)**.** *Let $\sigma \in \mathcal{C}(\mathbb{R}, \mathbb{R})$ be a non-affine continuous function differentiable with continuity at at least one point, with non-zero derivative at that point. Let $K \subseteq \mathbb{R}^n$ be a compact set. Then $\mathcal{N}_{n,m,\infty}^{n+m+2,\sigma}$ is dense in $\mathcal{C}(K, \mathbb{R}^m)$.*

So we have as well a justification for the case of arbitrary depth. In every practical case, we obviously cannot have arbitrary depth or width, but with these theorems it's reasonable to think that with an appropriate combination of both dimensions, the procedure explained previously may be a way to approximate the desired function.

These theorems also indicate that, in general, the more the depth and width, the better the performance of the model will be. This has been manifested in the current trend in artificial intelligence, in which networks are getting bigger. In particular, they are getting deeper, since some theorems show that depth is more important than when arbitrary dimensions are not considered. This is further discussed in [Dan17].

The references for this section have been [KL20], [Pin99] as well as Section 8.4 of [GBC16].

3.4 Convolutional Neural Networks

There are many architectures for neural networks, and one that has become of most importance is **convolutional neural networks**. This is so because of their great success in image recognition, classification and segmentation, among other applications in which the spatial structure of the data is important, such as time series or audio.

The structure of a convolutional neural network is the same as that of a fully connected neural network, but, at least one of its layers is what it's known as a **convolutional layer**. This layer is usually constituted by three stages. The first one is the **convolutional stage**. Let's say the input to layer $l+1$ is $I = (a_1^{(l)}, \dots, a_{n_l}^{(l)}) \in \mathbb{R}^{n_l}$. We define the **kernel** as a certain $K = (k_{-s}, \dots, k_s) \in \mathbb{R}^{2s+1}$ with $2s+1 \leq n_l$, which will describe the operation in the layer. For $1+s \leq i \leq n_l - s$ we will have that:

$$z_i^{(l+1)} = (K * I)_i = \sum_{m=-s}^{s} a_{i+m}^{(l)} \cdot k_m$$

Where this relation is not defined, that is the borders of the vector I, a possible strategy is to add a number of zeros or the average of the limits to increase the sides of I. This way, the limits of the input are take into account more than once. This is known as **padding**. If $2s+1 = n_l$ and we apply no padding, $z_i^{(l+1)}$ is the same as that of a fully connected layer with weights K and biases equal to zero. Notice that we will be using the same kernel, or a small set of kernels, for all the convolutions in the layer.

We can think of this convolutional operation as sliding the kernel along the input, and computing the corresponding operations. For the case of images or inputs with higher dimensions, the kernel will be also of the corresponding dimension. Thus, if we have a

picture, we can image the operation as sliding a squared kernel over a grid, covering all the image in the process. The sliding of the kernel needs not be of one pixel at a time, the chosen distance is known as **stride**. To simplify the notation we'll focus on vectors as inputs.

The second stage is the application of a non-linear activation function σ. Finally, there is the **pooling stage**. In it we reduce the number of components of the previous vector to the desired n_{l+1} obtaining the values of the layer, $(a_1^{(l+1)}, \ldots, a_{n_{l+1}}^{(l+1)})$. To achieve this, we combine consecutive entrances of the result in the previous stage, mainly with the average or picking the maximum. The number of consecutive values used for pooling depends on our choice.

There are some particularities that give this construction various advantages. First of all, usually $2s + 1$ is much smaller than n_l. This prevents far away input neurons to interact with each other, what might be of interest when considering structured data such as images. This, combined with pooling and the fact that a few kernels are used for all the convolutions, results in a drastic reduction of the number of parameters, a crucial factor when it comes to training models with a large number of inputs, such as big images. We may use local kernels if we are trying to detect different features in the same input.

Equivariance to translations is another consequence of convolution and pooling. This is due to the fact that a repeating kernel and then averaging or choosing the maximum may maintain the resulting value for small shifts of the input. This is interesting when we are interested in seeing whether or not a feature is present, rather than its precise location.

In convolutional neural networks, the last layer is a fully connected one. When the inputs have more than one dimension, the neurons have to be reshaped to a vector before entering the last layer, a process known as **flattening**. The design of the convolutional layers is not fixed, we may have some convolution and activation stages before pooling.

The references for this section have been week 2 of [GGW22], Chapter 9 of [GBC16] as well as [JZ21].

3.5 Local Feature Attribution

Deep neural networks have been successful in their application. However, the function they induce, $\tilde{f}$, is not very useful when it comes to understanding what the model is doing, since it is a long composition of very-large-dimension matrices. We would like to understand which features the model is regarding as essential. As we have said, this part has been crucial for the research at DeepMind. In this section we will briefly study a couple of ways to gain insight into the hidden layers. A more precise definition of what we mean when we say this is the following:

Definition 36. Let $\tilde{f} : \mathbb{R}^n \to \mathbb{R}^m$ be a function of the class of neural networks $\mathcal{N}^{\sigma}_{n,m,N}$. Let $k \in \{1, \ldots, m\}$. An **attribution vector** of the $k-$th prediction of the **explicand** $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$, $\tilde{f}_k(x)$, is a vector $A^k(x) = (a_1^k, \ldots, a_n^k) \in \mathbb{R}^n$, where each a_j^k is considered to be the **contribution** of x_j to the output $\tilde{f}_k(x)$. This defines a **feature attribution** map $A^k : \mathbb{R}^n \to \mathbb{R}^n$.

3.5.1 Perturbation-based Methods

As the name indicates, these methods alter parts of the input in order to generate explanations of the model. We will see how to apply Shapley values to the interpretation of models, focusing on the solution proposed by the framework SHAP.

First of all, we have to find a way to assign an $n-$player game in a coalitional form to our model $\tilde{f}$. Let say we want to analyze the input $x^{(0)} = (x_1^{(0)}, \ldots, x_n^{(0)}) \in \mathbb{R}^n$. Then, the players would be $x_1^{(0)}, \ldots, x_n^{(0)}$. Once we define the characteristic function v, we could apply the division of the payoff given by the Shapley values $(\phi_1(v), \ldots, \phi_n(v))$, that we know it is fair by Theorem 31. Each of the $\phi_k(v)$ would be interpreted as the contribution of each of the inputs to the difference between the particular prediction and the mean prediction in the concrete case the input is $x^{(0)}$. Nevertheless, we have to make a couple of considerations.

First, let's see how to define v. As we know, Shapley values are given by the marginal contribution of each coalition. This means that we should compute $\tilde{f}$ without a subset of the inputs, which at first glance does not make much sense. A way to do so would be by retraining the model with each subset of the inputs, resulting in a very slow process. Furthermore, in some cases such as image recognition, it doesn't make sense to train the network without all the inputs. A solution to solve this would be choosing a **baseline** sample $x^{(b)}$, and then defining $v(C) = \tilde{f}(\pi(x^{(0)}, x^{(b)}, C))$, where

$$\pi_k(x^{(0)}, x^{(b)}, C) = \begin{cases} x_k^{(0)} & \text{if } x_k^{(0)} \in C \\ x_k^{(b)} & \text{if } x_k^{(0)} \notin C \end{cases}$$

The choice of the baseline is pretty arbitrary, which may influence the results of the examination. This is why the **distributional replacements** are usually the choice. These consist on choosing the absent inputs from the probability distribution of the values in the sample. This distribution has to be approximated, and this is done by choosing a sample set $D_s \subseteq \{x^{(1)}, \ldots, x^{(M)}\}$ and then:

$$v(C) = \frac{1}{|D_s|} \sum_{x^{(b)} \in D_s} \tilde{f}(\pi(x^{(0)}, x^{(b)}, C))$$

The other problem we encounter using Shapley values is the complexity of the operations required, since the number of coalitions grows exponentially with the number of players. There are a number of approximating techniques, which are out of the scope of the project. Examples are KernelSHAP (not specific of any model) or DeepSHAP (specific of deep networks). Both use the marginal removal approach and will be employed in the next chapter. Regardless of this problem, the advantages of using Shapley values are its solid theoretical background and the fulfilling of the fairness definition.

3.5.2 Gradient-based Methods

These methods are based on investigating changes on the output as consequence of slights variations in the input.

To understand the concept, let's first consider the case where $\tilde{f} : \mathbb{R}^n \to \mathbb{R}^m$ is a linear function. In this case, we have that $\tilde{f}(x) = Wx + b$, where $W \in \mathbb{R}^n \times \mathbb{R}^m$ and $b \in \mathbb{R}^m$.

Then, $\tilde{f}_k(x) = w_{k,1}x_1 + \cdots + w_{k,n}x_n + b_k$. From here we can deduce that the contribution of each feature is proportional to the corresponding weight.

Now for the general case, let $x^{(0)} \in \mathbb{R}^n$ be the input we want to analyze. Assuming differentiability, we can approximate the function $\tilde{f}_k(x)$ for x near $x^{(0)}$ with:

$$\tilde{f}_k(x) \approx \tilde{f}_k\left(x^{(0)}\right) + \nabla \tilde{f}_k\left(x^{(0)}\right) \cdot ||x - x^{(0)}||$$

We will consider that the importance of $x_1^{(0)}, \ldots, x_n^{(0)}$ is proportional to the magnitude of the gradient. Backpropagation will be used to compute the derivatives. If we have $\nabla \tilde{f}_k\left(x^{(0)}\right) = (r_1, \ldots, r_n)$, we will define $A^k(x^{(0)})$ by $a_j^k = r_j$ or $a_j^k = |r_j|$.

There are different gradient-based methods, which mainly differ in the way the gradient is backpropagated. All of them are widely used in analysing images, and referred to as **pixel attribution** or **saliency maps**. This is the case because we can assign to each original pixel a new value according to the mapping defined above, visually representing its importance. The results of these methods should be taken with a grain of salt, and rather as a qualitative analysis, since inconsistencies in the results have been found between different gradient-based methods.

The references for this section have been Sections 9.5, 9.6, 10.1 i 10.2 of [Mol23] as well as [STY17], [SVZ14] and [CCLL22].

Chapter 4

Experimental Procedure

In the following sections we will develop a couple of examples where we applied machine learning to problems regarding group theory. Apart from getting the models to work, one of the goals will be to analyze the models through feature attribution methods. The code has been adapted from various examples found online, and a very useful learning resource has been a seminar organised by the Sydney Mathematical Research Institute, [GGW22].

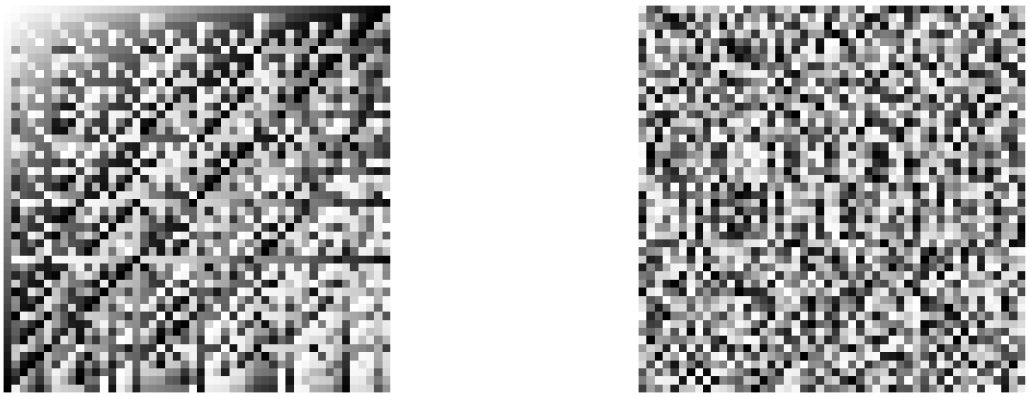
4.1 Image Classification

The Cayley table (Definition 6) gives us a way to represent a group, and a matrix is a very compelling object for a computer to work on. As it's shown in [HK19], where they use the Cayley table to investigate some properties, this way of describing groups has been successful in training some machine learning algorithms. However, they don't get into a feature attribution analysis. One area where machine learning has had great success is in image classification with the use of convolutional neural networks. Furthermore, gradient-based methods give us a way to, more or less, see what the models are regarding as important.

Combining these two considerations is how the idea of this section originated. The idea is to use the Cayley table to make a neural network learn a certain property, but instead of using the matrix representation, converting the table to an image. Then, we would like to make a saliency map of the inputs and see if this could help recognise areas of more importance and thus, providing useful mathematical information.

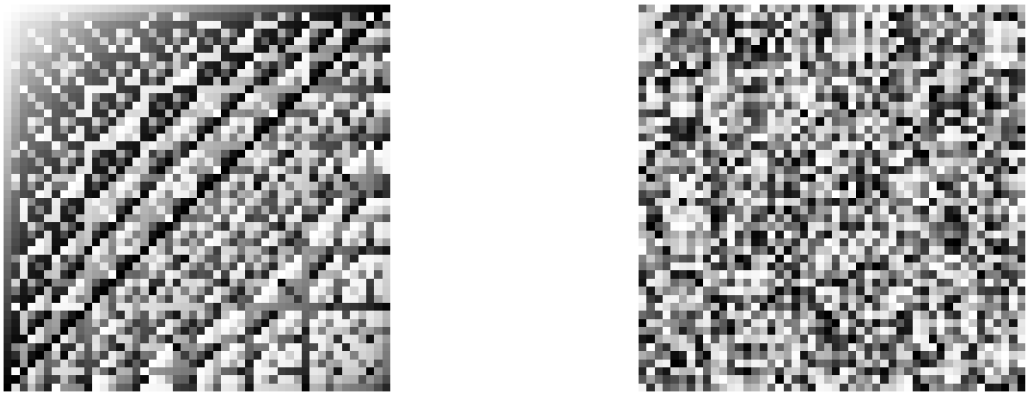
So, first of all, the specifications of a useful database had to be defined. For this, we had to consider groups of an order big enough that allowed different examples of each property, but for computational limitations, as well as properly visualising the attribution, we should consider rather small orders. In conclusion, after examining different options, the choice was focusing on groups of order 48, and classifying whether they are nilpotent or not (Definition 17). This is because there are 52 groups of order 48, 14 of which nilpotent. To compute the Cayley tables, as well as the characteristics of each group, the program GAP with the GAP Small Groups Library package has been used, [Gro22], [BEOH22]. For the rest of the computations, Python was the language of choice.

Afterwards, the images from the tables were created. For this, we assigned a different gray-scale color to each element. Obviously, we only have 52 groups, so we must increase the elements of the database. For this, we did permutations of rows and columns until we had both a balanced and a large dataset. That is, we did more permutations for each of the nilpotent groups than for each of the no nilpotent. There are many permutations of 48 elements, so repeated inputs won't be a problem. In total, the database consisted in 8100 nilpotent groups and 8170 no nilpotent groups. We can see an example of how the inputs look in Figure 4.1 and Figure 4.2. Classifying certain kind of images is really easy for us, but it's clear from looking at the pictures that this wouldn't be the case.



(a) Without permutations. (b) With permutations.

Figure 4.1: No nilpotent group, `SmallGroup(48,1)`.



(a) Without permutations. (b) With permutations.

Figure 4.2: Nilpotent group, `SmallGroup(48,2)`.

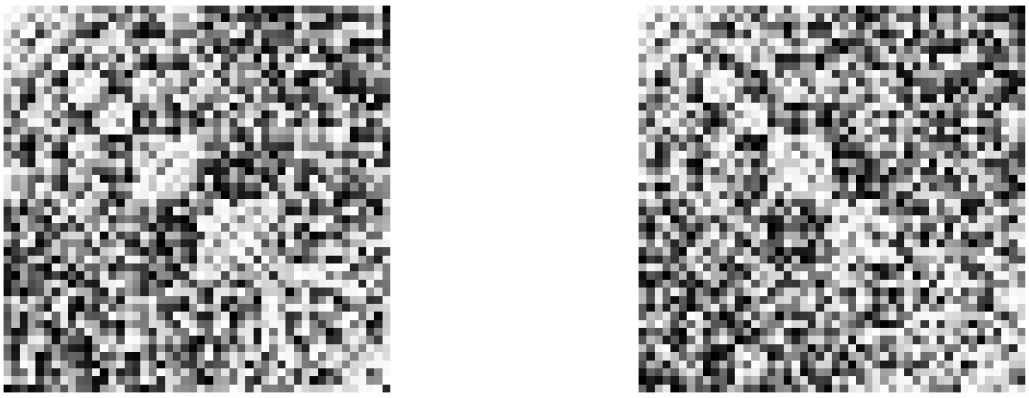
It's turn to see if a convolutional neural network can classify the images. For this, we trained the model with varying the batch size, number of epochs, and percentage of data allocated for training and validating. We got up to 90% of accuracy in the validation set training with only 20% of the database as the training set. If we increased the training set to 50%, the validation accuracy was more than $99,9\%$.

To investigate a little further, let's recall that in a nilpotent group, the nilpotency class can be defined (Definition 17). For nilpotent groups of order 48, there are three possible

options: 1, 2 and 3. The database consisted in 2700 examples of each, and we trained the network with only nilpotent groups to see if it could classify them in the class they correspond.

After trying with different combinations of the hyperparameters, the network didn't perform better than getting 1/3 of the validating data correctly, which is no better than random.

The elements in a group, can be ordered according to Definition 2. So we could try the following: arrange the elements in the Cayley table according to its order, and then, instead of doing random permutations, permuting only the elements of the same order. Again, the dataset consisted on 2700 inputs for each class. We can see a couple of examples of this dataset in Figure 4.3.



(a) Elements arranged by its order. (b) Permutation preserving elements' order.

Figure 4.3: Nilpotent group of nilpotency class 1, `SmallGroup(48,2)`.

So we retrained the model with this new database, and we obtained better results, about 80% of accuracy in the validating set. This is not a really high value, but it's a significant improvement. It should be further studied to understand what is the paper of convolution in this example, since it is not clear what are the consequences of this operation in our example, neither the differences between preserving the order or not.

The central question of all this is whether the network is learning something of mathematical relevance, or it just found a way to classify the images with pretty good results. The high accuracy when it comes to detecting nilpotency, even with low training percentages, indicates that the model generalizes fairly well and this could be thanks to finding mathematical patterns in the dataset.

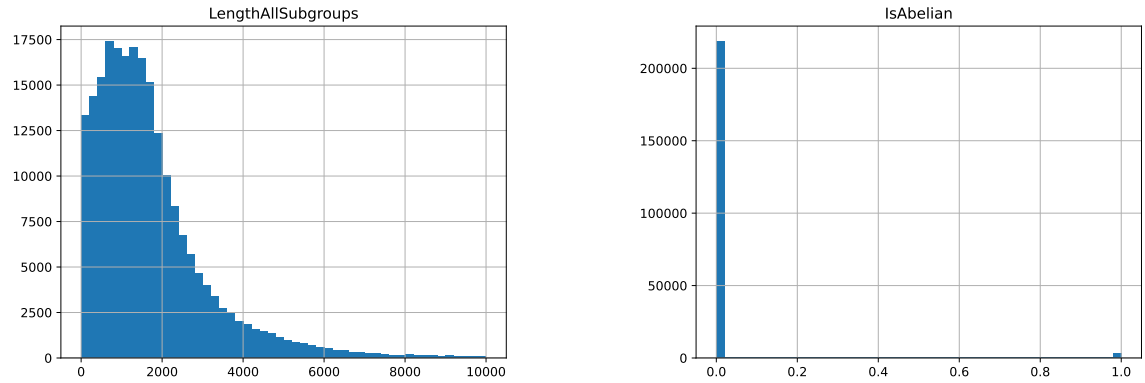
To check which of the regions of the picture trigger the decisions, the idea is to apply pixel attribution. Maybe we could then see that the model is only considering certain regions such as the diagonal, the borders or certain rows as relevant. Furthermore, we could compare the results of saliency for each of the possible outputs, and maybe find a consistent pattern among all the predictions. If this made any sense, even a conjecture could be made. Even though I have tried many times to program this attribution part, I haven't managed to succeed in making it work. The Python code can be found in Example 1.

4.2 Data Prediction

In this second example, we will see if we can find any interesting relationship among different group properties. To do so, we constructed a dataset that consists on 30 different properties for 221326 finite groups using the GAP software with the GAP Small Groups Library package, [Gro22] and [BEOH22]. This procedure is inspired by DeepMind's research regarding knot theory in which, as we have explained in the introduction, a large dataset of geometric invariants for knots was used to train a neural network that predicted the signature of the knots. We will try to see if we can predict one of the properties from the others, and then review which were the contributions of the inputs to the prediction, and see if the model is highlighting something of mathematical interest.

The database consists on binary properties such as whether the group is commutative, nilpotent, or a $p-$group, among others. It also includes non-binary data such as the nilpotency class, the number of the conjugacy classes, the number of subgroups, the number of normal subgroups or the orders of the group, the Frattini subgroup (Definition 22) and the Fitting subgroup (Definition 24). There are groups up to order 2000, even though not all are present. Some of the groups we excluded were those of orders 512 and 1024, since there are millions of those with the same properties, and thus the neural network would be inclined to only learn this fixed values. Others, such as what is known as minimal faithful permutation degree, are not present in the database due to computing and time limitations.

First of all, we made a preliminary analysis of the data through histograms to determine the distribution of the properties and see what made more sense to study, we find an example of these histograms in Figure 4.4.



(a) Histogram of the number of subgroups. (b) Histogram of commutative groups.

Figure 4.4: Previous study of the dataset.

The first conclusion was to train a model to detect if a group is nilpotent or not from the other properties. This was due to different reasons, one being that about a third of groups were nilpotent. This doesn't happen in other binary characteristics such as being commutative or being a $p-$group, since almost all groups are not, and the database would be too skewed for our purposes. Secondly, some non-binary operations were fairly nicely distributed to predict, but it is a more difficult task since there are less training examples

for each of the values. For instance, the number of normal subgroups ranges from 1 to 417199. This is the reason why we will start with this binary classification.

The next step was to compensate the number of nilpotent groups in the dataset to increase the proportion from a third to a half, so that half of the values correspond to nilpotent groups. With this, we trained the neural network getting a model $\tilde{f}$ with very high accuracy, more than 99%. The important part, however, is the analysis of the results, that was implemented through the perturbation-based feature attribution method given by the SHAP Python library. We computed the Shapley values of the different properties in the database for a random sample D_s of 2000 groups, and we plotted their mean values in Figure 4.5. We did this using the method given by KernelSHAP, and the sample wasn't bigger because of the large computation time required. With the notation used in the last chapter, we are computing $\phi_k(v)$ where k is one of the group properties and

$$v(C) = \frac{1}{2000} \sum_{x^{(b)} \in D_s} \tilde{f}(\pi(x^{(0)}, x^{(b)}, C)).$$

As Theorem 25 yields, the Fitting subgroup of a finite group is its largest normal nilpotent

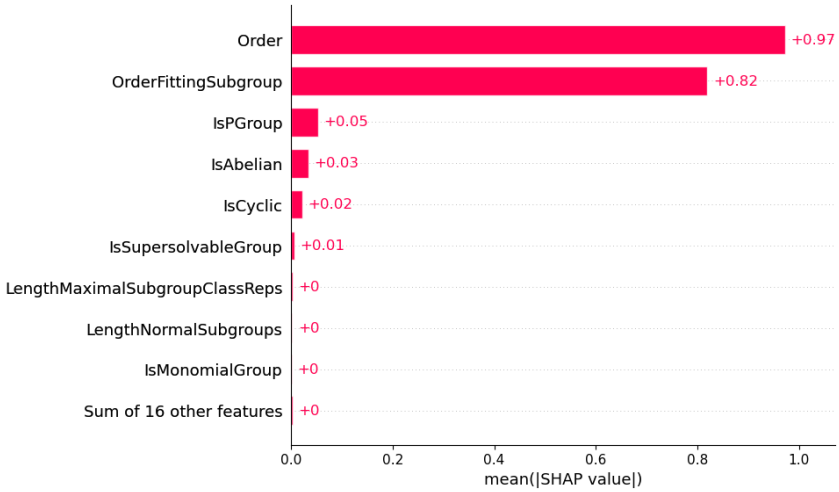


Figure 4.5: Mean absolute SHAP values for 2000 groups.

subgroup. This means that when the group is nilpotent, the order of the group is the same as the order of the Fitting group. Considering Figure 4.6, it is clear that the neural network has captured this relation and it is using it to predict the task at hand: that is because the two features are the more important, and this might mean the model is comparing them.

It is important to remark that this result follows almost directly from the definitions, and would have been also discovered by analyzing the correlations in the original database. However, it is still useful for two reasons. On the one hand, it corroborates the fact that the neural network is working and validates its accuracy. On the other hand, it verifies that the analysis done from the Shapley values is meaningful. This motivates developing further this example.

To do so, we retrained the model eliminating the order of the Fitting group from the input. Again we be obtained really good accuracy, more than 95% for the validation inputs. We

computed the Shapley values for 2000 groups and obtained the information of Figure 4.6. We can see that the predominant property is being or not a $p-$group (Definition 19). This

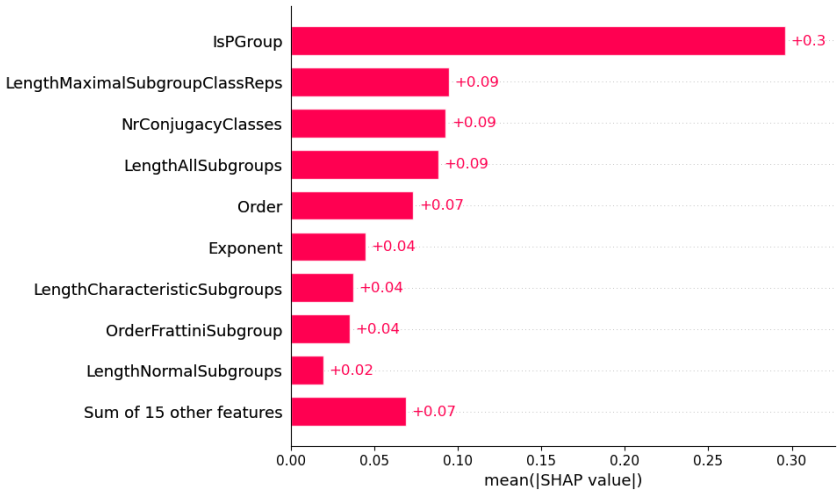


Figure 4.6: Mean absolute SHAP values for 2000 groups.

result is in accordance with Theorem 20, that states that finite $p-$groups are nilpotent.

So, similarly as we did before, we retrained the network without the last two properties (the order of the Fitting group and whether the group is a $p-$group or not) to see how would the model perform. We found that the model had about 93% of validation accuracy. The results of the SHAP application can be found in Figure 4.7. In the previous to

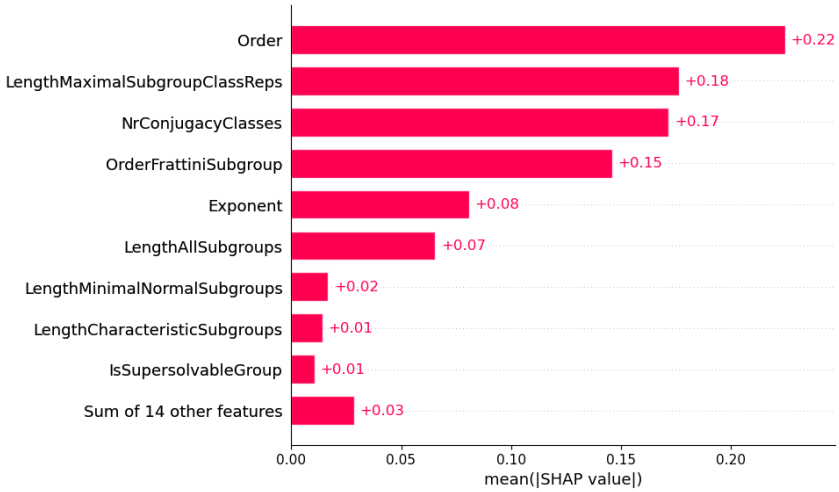


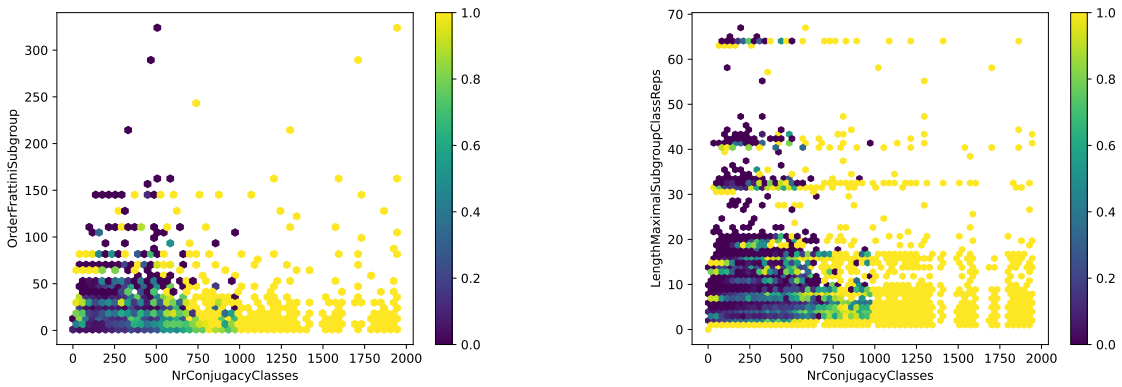
Figure 4.7: Mean absolute SHAP values for 2000 groups.

attempts, we had two prominent features, while now we have four that are more relevant. This might mean that the decision is not so obvious now that we removed the previously most relevant inputs. Moreover, notice that these four main properties also appeared in the previous figure at a second place, so the results seems to be consistent with the previous trainings.

We see that the most prominent features are the order of the group, the number of conjugacy classes (Definition 12), the number of conjugacy class representatives of maximal subgroups and the order of the Frattini subgroup (Definition 22).

In Google's collaboration, [DVea21], we see the relevance of machine learning, but only when combined with classical mathematical procedures. This is why after training the model and determining the most important features, they went back to the dataset to check the new relationships suggested by the network.

After analyzing different combinations among the four most relevant features, we will now discuss the ones that appeared to be of more relevance. First of all, let us take a look at Figure 4.8a. In it we plot the order of the Fitting subgroup with respect to the number of conjugacy classes. The color of each hexagon represents the proportion of nilpotent groups inside its area. That is, in a yellow hexagon all groups will be nilpotent, and in a purple none of the groups will be nilpotent. With a regular scatter plot there would have been too many overlapping points to see any pattern.



(a) Order of the Frattini subgroup, number of conjugacy classes.

(b) Number of maximal subgroups, number of conjugacy classes.

Figure 4.8: Previous study of the dataset.

We can compare this plot to the one in Figure 4.8b, and it makes sense that they are similar considering the definition of the Frattini subgroup.

Finally, we plotted the number of conjugacy classes with respect to the order of the group, and we can see the results in Figure 4.9. This plot shows that there seems to be a relevant correspondence among the order, the number of conjugacy classes and whether the group is nilpotent or not. Groups in the diagonal are commutative groups, so the closer a group is to this line, the closer it is to being commutative. Notice that there is a last purple line, and above it, all groups appear to be nilpotent. There is also a yellow line between all the purple points, which may be interesting to further study.

Let G be a finite group and $\kappa(G)$ the number of conjugacy classes of G as defined in Definition 12. From this plot we could maybe conjecture that if G is nilpotent, then $\kappa(G) > C|G|$, for some $C > 0$. We could expand the database to include more examples, and maybe adjust the hypothesis accordingly to find a more fine bound for $k(G)$. We

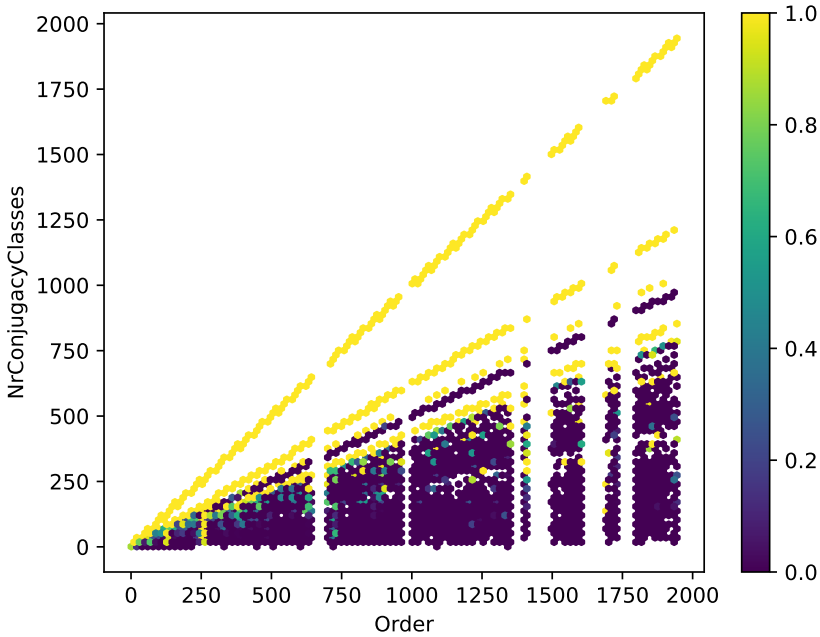


Figure 4.9: Number of conjugacy classes, order.

could even investigate which is the slope C of the last purple line, supposing it is a line. In reference [JZ11, Thm. 1.1], we see that there have been investigations related to this topic. For example, there we can find the following theorem:

Theorem 37. *There exists a explicitly computable constant $C > 0$ such that every finite nilpotent group G of order $|G| \geq 8$ satisfies*

$$\kappa(G) > C \frac{\log_2 \log_2 |G|}{\log_2 \log_2 \log_2 |G|} \cdot \log_2 |G|$$

Moreover, the following conjecture is proposed:

Conjecture 38. *There exists a constant $C > 0$ such that a finite group G satisfies $k(G) \geq C \log_2 |G|$.*

So, a conjecture of this kind was already proposed by other mathematical investigations. Another reference in this topic is [AHADaAA17].

Now, we will try now to identify the yellow lines above the last purple line in Figure 4.9, as well as this last purple line. It is clear that the elements in the diagonal should be commutative groups, and it has been confirmed checking the database that this is the case.

The next line has a slope of 5/8, which corresponds to groups of order 8 and 5 conjugacy classes. It is known that the groups Q_8 and D_4 satisfy these properties, and they lay in this line. Inspecting the database confirms that this are the first points with this slope. Moreover, Proposition 14 gives us a way to build more groups in this line. If G_1, G_2 are

groups and G_2 is commutative, we have that:

$$\frac{\kappa(G_1 \times G_2)}{|G_1 \times G_2|} = \frac{\kappa(G_1) \cdot \kappa(G_2)}{|G_1| \cdot |G_2|} = \frac{\kappa(G_1)}{|G_1|}$$

So a way of constructing groups in this line would be by considering the product of Q_8 or D_4 with a commutative group. However, not all groups in this line are of this form. For example, `SmallGroup(64,3)` is in the line and is not of this form.

The following line has a slope of 17/32 which could correspond to groups of order 32 and 17 conjugacy classes. Taking a look into the database, we have that there are two groups with this properties:

$$A = \texttt{SmallGroup(32, 49)} \qquad B = \texttt{SmallGroup(32, 50)}$$

Then we find two purple lines of slope 1/2 and 65/128, which are very close to each other. Studying the database we ensured that this five slopes are the only slops greater or equal to 1/2. However, we have to remember that the database has only groups under order 2000, so other lines may appear at higher order.

Finally, there is a yellow line between the purple region which has a slope of 7/16. The first two groups in the database in this line are Q_{16} and D_8. All these has to be seen as a curiosity and an open question that could be further investigated, rather than a rigorous study.

I would like to stress the following point. Clearly, all these last plots could have been made from the beginning, and maybe inspecting directly the database would have been quicker. However, we are presenting these results as the blueprint for the application of artificial intelligence to mathematics, following the steps from the team at DeepMind and their mathematician collaborators. For example, relating the number of conjugacy classes to the order and nilpotency is pretty sensible considering the definitions, but in more complex cases, the feature attribution could throw light to less obvious relationships. I believe the important point is having checked that both the neural network gives reasonable results and the posterior analysis using SHAP gives mathematically logical results. The code for this example can be found in Example 2.

4.3 Other Attempts

Sections 4.1 and 4.2 are the last steps of a learning process that started with the examples and problems at [GGW22], that were really useful to start understanding how are neural networks actually defined and trained. After this preparation, I played around a little bit with the code used in DeepMind's collaboration, [DVea21]. It is more sophisticated and it was my first contact with a feature attribution method. I observed how the importance of the features changed when altering the inputs present in the database, similarly in the second example.

Having practiced a little bit, I started to think of an example of where to apply all this. The original idea was to make a neural network that differentiates trivial or non-trivial groups given what is known as the presentation of the group. The main problem

was the database, since there is only one trivial group, and getting thousands of different presentations of it was difficult. I managed to make a network that classified some groups, but it was only useful as a practice. Furthermore, feature attribution could not have been made since it does not make sense to directly compare presentations of different groups. This is because presentations have different lengths, the elements may be repeated or superfluous: there is not a rigorous way to define what the inputs are in contrast to, per example, having 30 defined group properties. It was this last consideration that resulted in discarding the example, since being able to analyze the model had to be an important part of the practical example.

I also tried to predict the order of the certain groups from their presentation, but this example did not even work in simple cases such as the dihedral group.

Chapter 5

Conclusion

5.1 Future Developments

Regarding the first example, I think it would be very interesting to make the pixel attribution work. This could give insight into what are the parts of the Cayley table that have been relevant to make the decision, and maybe this would result in interesting patterns. Even though a precise information cannot be deduced from this analysis, comparing the qualitative results of different methods could be relevant. For example, we could check if there is any difference between the relevant features when the Cayley table is ordered and when it's not. This could not only be of mathematical interest, but it could be a way of understanding a little bit more about convolutional neural networks. Finally, the case were groups are classified according to its nilpotency class should be tuned to see if the 80% accuracy can be improved.

The second example provides different opportunities. First of all, there could be a more thorough mathematical review of the relationship shown in Figure 4.9, trying to understand which groups constitute each line, and increasing the number of groups of the dataset. Also, taking advantage of all the different properties available in the dataset, it could be interesting to see if other models could make accurate predictions. In particular, it would be interesting to try to predict a non-binary function, rather than classifying the groups in two classes. This is challenging but worth further developing.

5.2 Conclusions

It is clear that machine learning is having immense impact in the current world with its practical applications. The main goal of this project was if these methodology could be transferred to theoretical endeavours, and propose ways artificial intelligence could help guiding mathematical intuition. This methods have been proposed in the two examples based on previous reports in literature. Even though I did not manage to apply pixel attribution analysis, the described method can hopefully be developed in future attempts. The analysis using SHAP was successful, and besides giving coherent results, it allowed to link game theory and Shapley values to machine learning, which is interesting.

Bibliography

[AHADaAA17] Bilal Al-Hasanat, Awni Al-Dababseh, Al-Sarairah andSadoon Alobiady, and Mahmoud Bashir Alhasanat. An upper bound to the number of conjugacy classes of non-abelian nilpotent groups. 2017.

[BEOH22] H. U. Besche, B. Eick, E. O'Brien, and M. Horn. *The GAP Small Groups Library, Version 1.5.1*, 2022.

[CCLL22] Hugh Chen, Ian C. Covert, Scott M. Lundberg, and Su-In Lee. Algorithms to estimate shapley value feature attributions, 2022.

[CMZ18] Anthony E. Clement, Stephen Majewicz, and Marcos Zyman. *The Theory of Nilpotent Groups*. Birkhäuser, 2018.

[Coh12] P.M. Cohn. *Basic Algebra: Groups, Rings and Fields*. Springer London, 2012.

[Dan17] Amit Daniely. Depth separation for neural networks. 2017.

[DFO20] A. Deisenroth, Aldo Faisal, and Cheng Soon Ong. *Mathematics for Machine Learning*. Cambridge University Press, 2020. https://mml-book.com.

[DVea21] Alex Davies, P. Veličković, and L. Buesing et al. Advancing mathematics by guiding human intuition with ai. *Nature*, 600(7887):70–74, 2021.

[Fer00] Thomas S. Ferguson. Game theory, class notes for math 167. 2000.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[GGW22] Joel Gibson, Georg Gottwald, and Geordie Williamson. Machine learning for the working mathematician, 2022.

[Gro22] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.12.2*, 2022.

[HK19] Yang-Hui He and Minhyong Kim. Learning algebraic structures: Preliminary investigations, 2019.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, 2015.

[JZ11] Andrei Jaikin-Zapirain. On the number of conjugacy classes of finite nilpotent groups. *Advances in Mathematics*, 227(3):1129–1143, 2011.

[JZ21] Shengli Jiang and Victor M. Zavala. Convolutional neural nets: Foundations, computations, and new applications. *CoRR*, 2021.

[KL20] Patrick Kidger and Terry Lyons. Universal Approximation with Deep Narrow Networks. In *Proceedings of Thirty Third Conference on Learning Theory*, pages 2306–2327, 2020.

[Mol23] Cristoph Molnar. *Interpretable Machine Learning*. 2023. https://christophm.github.io/interpretable-ml-book/index.html.

[Nie15] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[Pin99] Allan Pinkus. Approximation theory of the mlp model in neural networks. 1999.

[San18] Grant Sanderson. Neural networks, 2018. https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi.

[Sha52] L. S. Shapley. A value for $n-$person games. *The RAND Corporation*, 295, 1952.

[STY17] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. *CoRR*, 2017.

[SVZ14] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. 2014.