University of São Paulo
Institute of Mathematics and Statistics
Bachelor of Computer Science

A study on linear piece wise approximation of Neural Networks

João Felipe Lobo Pevidor

Final Essay

mac 499 — Capstone Project

Supervisor: Prof. Dr. Marcelo Finger
Co-supervisor: Dr. Sandro Márcio da Silva Preto

São Paulo
2022

Resumo

João Felipe Lobo Pevidor. **Um estudo sobre aproximação de Redes Neurais por funções lineares por partes**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

Redes Neurais são conhecidos como modelos de caixa preto em inteligência artificial, isso significa que não podemos acessar diretamente a informação que o modelo utiliza para operar ou justificar suas decisões. Ultimamente, tem existido um grande interesse em pesquisa sobre isso e múltiplas técnicas estão sendo desencolvidas tentar abordar esse problema. Particularmente, Sandro Preto na sua tese de doutorado Preto, 2021 descreve um método que foi desenvolvido para atacar esse problema e apresenta um algoritmo para realizar a inferência formal de propriedades de uma dada rede neural. O único porém desse método é que sua entrada deve ser uma aproximação da rede, na forma de uma função linear por partes. O objetivo desse trabaho é estudar e pesquisar métodos para gerar essas aproximações e complementar o trabalho realizado na tese mencionada.

Palavras-chave: Redes Neurais. Lógica de Lukasiewicz. Modulo satisfabilidade. Funções lineares por partes. Aproximação.

Abstract

João Felipe Lobo Pevidor. **A study on linear piece wise approximation of Neural Networks**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

Neural Networks are famously known as black box models in artificial intelligence, that means that we cannot directly access the information on which it operates or justify its decisions. Lately, there has been a big interest in research and multiple techniques are being developed to try and approach this problem. In particular, Sandro Preto in his thesis Preto, 2021 describes a method developed to help tackle this problem and presents an algorithm to formally infer certain properties about a given network. The only thing is that it relies on the existence of an approximation of a Neural Network using linear piecewise functions. The aim of this work is to study methods to generate this approximation and complement the work done in the previously mentioned thesis.

Keywords: Neural Networks. Lukasiewicz Logic. Modulo Satisfability. Piecewise linear functions. Approximation.

List of abbreviations

MIP	Mixed Integer Linear Programming
ReLU	Rectfied Linear Unit
TId	Truncated Identity function
MLP	Multi layered perceptrons

List of symbols

$\hat{f}$	An estimator for function f
$Ł_\infty$	The logical system of Łukasiewicz Infinitely-valued Logic
$\mathbb{P}$	Set of propostional variables
Γ	Set of logical connectives
$\mathcal{L}$	A logical language
v	Valuation function
α, β	Propositional variables
$\neg$	$Ł_\infty$ negation
$\vee$	$Ł_\infty$-maximum
$\wedge$	$Ł_\infty$-minimum
$\rightarrow$	$Ł_\infty$-implication
$\oplus$	$Ł_\infty$-disjunction
$\odot$	$Ł_\infty$-conjunction

List of Figures

List of Tables

Contents

,

Chapter 1

Introduction

Neural Networks are famously known as black box models in artificial intelligence, which means that we know little to nothing about what happens inside of them. A block box model is a system which inputs and outputs are observable, but there is no knowledge available of its inner workings. To analyse behaviour of such models one usually has to observe the response generated by certain inputs and try to infer the model's machinery. Lately, there has been a big interest in research and multiple techniques are being developed to try and understand more about what happens inside of these models.

In particular, Preto (2021), has developed a method to help tackle this problem and has presented an algorithm to formally infer certain properties about the network by using Łukasiewicz Logic. This method relies on two important facts. The first one is that, as shown by S. Aguzzoli and D. Mundici, 2001 Stefano Aguzzoli and Daniele Mundici, 2003, for any continuous function $f : [0,1]^n \to [0,1]$ there is a linear piece wise function called rational McNaughton function $\hat{f} : [0,1]^n \to [0,1]$ such that

$$|f(x) - \hat{f}(x)| < \epsilon, \epsilon > 0.$$

And the second is that, any rational McNaughton function $\hat{f}$ can be expressed as a logical formula in Łukasiewicz Logic. With that, the algorithm presented by Preto allows us to make logical inferences from a trained model.

The process of verification of neural network consists in a four part process that is shown in 1.1

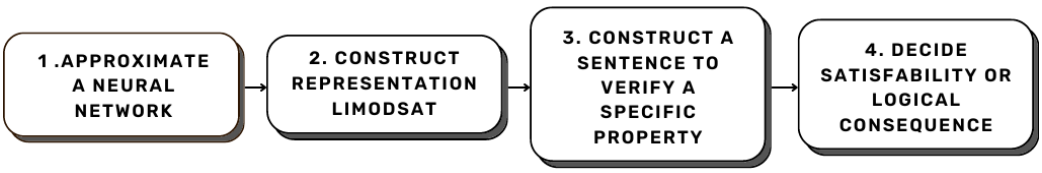

Figure 1.1: *Process of verification of a neural network*

To perform the formal verification of a neural network we need to combine multiple theoretical results and techniques. In the next chapters (2, 3, 4) we will introduce important

concepts (such as Łukasiewicz Logic and Neural Networks) and the theoretical results we need to be able to this.

In chapter 5, we introduce a known dynammic programming algorithm to approximate such models with a linear piece wise function which is what was used in the experiments detailed after.

Chapter 6, details issues found during the process of experimentation and how they were circumvented in order to be able to produce results.

In chapter 7, we detail the experiments and show the results obtained with the approximation of a neural network that approximate the XOR boolean function.

Finally, in chapter 8 we draw some conclusions about the process of verifying neural networks and some comments about the path taken to do so in this work.

Chapter 2

Łukasiewicz Logic

2.1 Classical Propositional Logic *vs* Łukasiewicz Logic

Classical Propositional Logic was created with the goal of being able to evaluate a statement systematically, in order to clearly determine if what was being presented was true or false. For that, mathematicians developed a system where you can represent and connect sentences (which are formally defined as propositional symbols or a composition of them) to create whatever statement possible, and of course, evaluate it. In this type of logic, sentences can only be evaluated as being true or false. On the other hand, we have logic systems that are called *many-valued logics* where it is possible to have a sentence evaluated to more than just 1 or 0, which are extensions of their classical peer. In this paper, we care about Łukasiewicz (infinetly-valued) Logic ($Ł_\infty$), where sentences can be evaluated by any real number between $[0, 1]$.

2.2 The $Ł_\infty$ language

To be able to work with such a system we, first, have to define the elements that we will be using, the $Ł_\infty$ language.

Definition 1. *Let $\mathbb{P}$ be an infinitely countable set of elements called propositional variables.*

Definition 2. *Let Γ be a finite set that contains all logical connectives.*

Initially it is sufficient to have the set $\Gamma = \{\neg, \rightarrow\}$, that will be extended later. Now we define the language inductively:

Definition 3. *Let $\mathcal{L}$ be a set such that:*

- *All elements of $\mathbb{P}$ are elements of $\mathcal{L}$;*
- *If $\alpha \in \mathcal{L}$, then $\neg\alpha \in \mathcal{L}$.*
- *If $\alpha, \beta \in \mathcal{L}$, then $\alpha \rightarrow \beta \in \mathcal{L}$*

$\mathcal{L}$ is referred to as the language of the system and because of the way it is defined, it contains all possible well-formed formulas.

2.3 Valuation function

To evaluate a formula, first we define a valuation function.

Definition 4. *Let $\mathbb{P}$ be the set of propositional symbols present in a logical system. A valuation of formulas $v : \mathcal{L} \to [0, 1]$ is a function that satisfies both conditions:*

$$1)\ v(\alpha \to \beta) \underset{def}{=} min(1, 1 - v(\alpha) + v(\beta))$$

$$2)\ v(\neg\alpha) \underset{def}{=} 1 - v(\alpha)$$

2.4 Ł$_\infty$ Connectives

In Łukasiewicz Logic, the Łukasiewicz implication: $\to$ and the negation: $\neg$ can be used as primitive connectives, and with them it is possible to derive other connectives and extend the set Γ. Using what was defined before, we can define the connectives $\vee$ and $\wedge$ in terms of the implication and the negation.

Definition 5. *For any two formulas $\alpha, \beta \in \mathcal{L}$, we define $\vee$ and $\wedge$ as:*

$$\alpha \vee \beta \underset{def}{=} (\alpha \to \beta) \to \beta$$
$$\alpha \wedge \beta \underset{def}{=} \neg(\neg\alpha \vee \neg\beta)$$

First, let's look at $\vee$:

Lemma 1. *Let v be the valuation previously discussed. Then:*

$$v(\alpha \vee \beta) = max(v(\alpha), v(\beta))$$

Now, we'll do the same for $\wedge$

Lemma 2. *Let v be the valuation previously discussed. Then:*

$$v(\alpha \wedge \beta) = min(v(\alpha), v(\beta))$$

With what we defined before, it is possible to compute the maximum and minimum semantic values. We can also define other connectives in order to develop more tools for arithmetic operations, such as the strong conjunction $\odot$ and the strong disjunction $\oplus$.

Definition 6. *For any two formulas $\alpha, \beta \in \mathcal{L}$, we define $\odot$ and $\oplus$ as:*

$$\alpha \odot \beta \underset{def}{=} \neg\alpha \rightarrow \beta$$
$$\alpha \oplus \beta \underset{def}{=} \neg(\neg\alpha \oplus \neg\beta)$$

As we did before, we can derive an expression for the valuation of these two from the implication and the negation. For $\oplus$:

Lemma 3. *Let v be the valuation previously discussed.*

$$v(\alpha \oplus \beta) = min(1, v(\alpha) + v(\beta))$$

And for $\odot$:

Lemma 4. *Let v be the valuation previously discussed.*

$$v(\alpha \odot \beta) = max(0, v(\alpha) + v(\beta) - 1)$$

Looking at these two connectives, we can see that with $\odot$ we can compute how much the sum of two semantic values surpasses 1. On the other hand, with $\oplus$ we can compute the truncated sum of two semantic values, which returns 1 if the result is greater than 1.[1]

2.5 Ł∞ satisfability

Note that many valuation functions can be created, that yield different results to the same formula, while still maintaining the two essential properties.

Definition 7. *We say that a formula ψ is Ł∞-satisfiable if there in a valuation v such that $v(\psi) = 1$. Otherwise, we say that ψ is unsatisfiable.*

We can extend this definition and say that a set of formulas Φ is satisfiable if, for some valuation function v, $v(\varphi) = 1$, for every $\varphi \in \Phi$.

Definition 8. *If for every valuation v, we have $v(\varphi) = 1$, we say that φ is Ł∞-valid.*

2.6 McNaughton Functions

Having introduced the fundamentals of Łukasiewicz Logic, we move to introducing another vital component to formally analysing a neural network McNaughton functions.

Definition 9. *A McNaughton function is a continuous function $f : [0,1]^n \rightarrow [0,1]$ for which there are linear polynomials $p_1, p_2, ..., p_m$ over $[0,1]^n$ with integer coefficients in a way that,*

[1] The proofs for every stated lemma can be found at Appendix A

for each $x \in [0,1]^n$, there is an index i in $\{1,...,m\}$ such that $f(x) = p_i(x)$. The polynomials $p_1, p_2, ..., p_n$ are called the linear pieces of f.

In McNaughton, 1951 McNaugthon showed that

Theorem 1. *For any McNaughton function $f : [0,1]^n \to [0,1]$, there is a logical formula $S(p_1, p_2, , ..., p_n)$ in Łukasiewicz Logic, such that $v(S) = f$, where p_i are sentential variables such that $v(p_i) = x_i$.*

2.6.1 Rational McNaughton Functions

For our purposes we are mainly concerned with rational McNaughton functions, which are a variation of the one previously mentioned

Definition 10. *A rational McNaughton function is a generalized function as in deinition 9 but whose linear pieces are allowed to have rational coefficients.*

Unfortunaltely, McNaughton (1951) also shows that only (integer) McNaughton functions may be represented by formulas of Łukasiewicz Logic. We may circumvent such situation by employing an implicit representation called representation modulo satisfability [Finger and Preto 2020; Preto and Finger 2020, 2022]. For that, first denote the set of formulas and denote by **Val**. Then, let Φ be a set of formulas and denote by $\mathbf{Val}_\Phi \subseteq \mathbf{Val}$ the set of valuations that satisfy Φ.

Definition 11. *Let φ be a formula and let Φ be a set of formulas. We say that a set of propositional variables $\mathbf{X}_n$,* determines φ modulo Φ-satisfiable *if:*

- *For any $\langle x_1, ..., x_n \rangle \in [0,1]^n$, there exists at least one valuation $v \in \mathbf{Val}_\Phi$, such that $v(X_j) = x_j$, for $j = 1, ..., n$;*
- *For any pair of valuations $v, v' \in \mathbf{Val}_\Phi$ such that $v(X_j) = v'(X_j)$, for $j = 1, ..., n$, we have that $v(\varphi) = v'(\varphi)$ - i.e. valuations in $\mathbf{Val}_\Phi$ are truth-functional on variables in $\mathbf{X}_n$.*

Definition 12. *Let $f : [0,1]^n \to [0,1]$ be a function and $\langle \varphi, \Phi \rangle$ be a pair where φ is a formula and Φ is a set of formulas. We say that φ* represents *f* modulo Φ-satisfiable *or that $\langle \varphi, \Phi \rangle$* represents *$f$* in the system Ł$_\infty$-MODSAT) *if:*

- *$\mathbf{X}_n$ determines φ modulo Φ-satisfiable;*
- *$f(v(X_1), ..., v(X_n)) = v(\varphi)$, for all $v \in \mathbf{Val}_\Phi$.*

Chapter 3

Solving a Satisfability Problem in Łukasiewicz Logic

It is known that it is possible to reduce the Satisfability Problem to a Mixed Integer Programming problem which is a generalization of a Linear Programming Problem.

3.1 Linear Programming

Linear Programming is a mathematical technique to maximize or minimize a linear function, subject to certain constraints. Given a certain function f, we want to maximize (or minimize) f in relation to all n-dimensional column vectors $x = (x_0, x_1, ..., x_n)$, subject to certain constraints. Our goal then, is to find $x^* \in \mathbb{R}^n$ such that $f(x^*) \geq f(x)\ \forall x \in \mathbb{R}^n$ (or $-f(x^*) \leq -f(x)\ \forall x \in \mathbb{R}^n$). This function, as the name of the technique suggests, has to be a linear function, so we define a n-dimensional column vector of costs $c = (c_0, c_1, ..., c_n)$ such that we can represent f as

$$f(x) = c^T x.$$

To construct the constraints, we can follow the same logic and have vectors a_i and scalars b_i represent the i-th constraint of the problem

$$a_i^T x \leq b_i, i \in A$$

$$a_i^T x = b_i, i \in B$$

$$a_i^T x \geq b_i, i \in C.$$

We may also constraints to specific elements of x:

$$x_j \leq 0, j \in D$$

$$x_j \geq 0, j \in E.$$

It is possible to further manipulate this formulation of the problem to create what is called the canonical representation of a Linear Programming problem. First, note that all equality constraints $a_i^T x = b_i$ can be turned into the pair of inequalities $a_i^T x \leq b_i$ and $a_i^T x \geq b_i$. Also,

all $a_i^T x \leq b_i$ constraints can be transformed into $-(a_i)^T x \geq b_i$. And finally, all constraints like $x_i \geq 0$ are special cases of $a_i^T x \geq b_i$ where all elements in a_i are 0, but the j-th.

Next, suppose that our problem has m constraints, let's define a matrix A such that

$$A = \begin{bmatrix} - & a_0^T & - \\ & \vdots & \\ - & a_m^T & - \end{bmatrix},$$

and the column vector $b = (b_0, b_1, ..., b_m)$. With the matrix A, the vector b and what was pointed out before, it is possible to write the whole linear programming problem as

$$\text{minimize } c^T x$$

$$\text{subject to } Ax = b$$

$$x \geq 0$$

where $A \in \mathbb{R}^{n \times m}$ and $c, x, b \in \mathbb{R}^n$.

3.2 Mixed Integer Linear Programming

Given the definition above, we can further constrain it as we please. One special case that is used to solve satisfability problems is Mixed Integer Linear Programming, which can be formulated as

$$\text{minimize } c^T x$$

$$\text{subject to } Ax = b$$

$$x \geq 0$$

$$x_i \in \mathbb{Z}, \forall i \in \mathbb{I}$$

where $\mathbb{I} \subseteq \{1, ..., n\}$.

3.3 Solving a Satisfability Problem in Ł$_\infty$

Hähnle showed in Hähnle, 1994 that the satisfability of a formula in Łukasiewicz Logic can be expressed as a Mixed Integer Programming Problem. In the article, a generalized tableaux procedure was introduced that made it possible to extend the use of the technique to many-valued logics, and infinitely valued logic. Making use of the concept of signed formulas and constraint rules, which were introduced in the paper as a means to extend the tableaux procedure, it is possible to verify if a certain formula is Ł$_\infty$-satisfiable. The constraints that result from a tableaux such as this can be interpreted as the feasibility part of a MIP problem and thus solving it would be, in fact, the same as checking the satisfability of the initial formula and is defined as the MIP problem associated with a certain formula.

A MIP-based solver for satisfiability in Łukasiweicz Logic would inductively translate a given formula ϕ to a set of MIP restrictions according to the rules for Ł$_\infty$ operators in

Table 3.1; each subformula ψ of ϕ is associated to a MIP variable x_ψ. A valuation v such that $v(\phi) = x_\phi$ may be derived from a feasible solution to the MILP restrictions.

Formula	**Restrictions**
X	$0 \leq x_X \leq 1$
$\neg\phi$	$x_{\neg\phi} = 1 - x_\phi$
$\phi \oplus \psi$	$b \in \{0, 1\}$ $b \leq x_{\phi\oplus\psi} \leq 1$ $x_\phi + x_\psi - b \leq x_{\phi\oplus\psi} \leq x_\phi + x_\psi$
$\phi \odot \psi$	$b \in \{0, 1\}$ $0 \leq x_{\phi\odot\psi} \leq b$ $x_\phi + x_\psi - 1 \leq x_{\phi\odot\psi} \leq x_\phi + x_\psi - b$
$\phi \vee \psi$	$b \in \{0, 1\}$ $x_\phi \leq x_{\phi\vee\psi} \leq x_\phi + b$ $x_\psi \leq x_{\phi\vee\psi} \leq x_\psi + 1 - b$
$\phi \wedge \psi$	$b \in \{0, 1\}$ $x_\phi - b \leq x_{\phi\wedge\psi} \leq x_\phi$ $x_\psi - (1 - b) \leq x_{\phi\wedge\psi} \leq x_\psi$
$\phi \to \psi$	$b \in \{0, 1\}$ $b \leq x_{\phi\to\psi} \leq 1$ $1 - x_\phi + x_\psi - b \leq x_{\phi\to\psi} \leq 1 - x_\phi + x_\psi$
$\phi \leftrightarrow \psi$	$b \in \{0, 1\}$ $1 - x_\phi + x_\psi - 2b \leq x_{\phi\leftrightarrow\psi} \leq 1 - x_\phi + x_\psi$ $-1 + x_\phi - x_\psi + 2b \leq x_{\phi\leftrightarrow\psi} \leq 1 + x_\phi - x_\psi$

Table 3.1: *Rules for building MILP Restrictions*

Given a set of formulas Φ, the solver decides on the feasibility of the MIP restrictions generated from Φ plus restrictions $x_\phi = 1$, for all $\phi \in \Phi$; Φ is satisfiable if, and only if, such restrictions are feasible.

Chapter 4

Neural Networks

4.1 Artificial Neural Networks

Neural Networks, or Artificial Neural Networks (ANNs) as they are called, are a category of models developed by the field of Machine Learning and are the foundations of today's deep learning algorithms that seem perform spectacularly in tasks such as face recognition, translation, among others. The inspiratin for ANNs is an algorithm called *perceptrons*, a linear model that implements a learning algorithm for pattern recognition, introduced by Frank Rosenblatt, 1958.

Mathematically, we can describe the perceptron as a weighted sum of its inputs. Let $X = (x_1, x_2, ..., x_n)$ be our input signals and $W = (w_1, w_2, ..., w_n)$ be a set of weights which will be used to evaluate the input. The algorithm consists, if we look at X and W as vectors, in the dot product between X and W followed by a threshold function f_φ.

$$X = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, W = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}$$

$$R = W^T.X = \sum w_i.x_i$$

$$f_\varphi(R) = \begin{cases} -1, & R < \varphi \\ 1, & R \geq \varphi \end{cases}$$

We can eliminate the need for φ introducing new coordinates to X and W:

$$X = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$$

$$W = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix}$$

and outputing a response with the following function f:

$$f(R) = \begin{cases} -1, & R < 0 \\ 1, & R \geq 0 \end{cases}$$

In order for the algorithm to output the correct response, the proper weights must be adjusted somehow. In the perceptron that will happen through trial and error. By presenting the algorithm with multiple input vectors $X = (X_1, X_2, ..., X_N)$ and a vector of correct responses for each signal $y = (y_1, y_2, ..., y_N)$, for each $X_i \in X$ we calculate

$$R_i = W^T.X_i$$

$$f(R_i) = \begin{cases} -1, & R_i < 0 \\ 1, & R_i \geq 0 \end{cases}$$

and update the weights according to a rule that tells us how to iteratively learn the correct response based on the lastest outcome. For the perceptron, that entails the following update rule:

$$\text{if } sign(W^T.X_i) \neq y_i \text{ then: } W = W + y_i.X_i$$

where W is the aforementioned weights vector, X_i is the i-th input signal presented and y_i is the correct response for that signal. The pseudocode for the perceptron algorithm can be written as:

Algorithm 1 Perceptron Learning Algorithm

Input
- X Data matrix of points
- y Data matrix of expected outputs for points in X

Output
- Hyperplane that separates the data

$W \leftarrow 0$

for each input example (X_i, y_i) **do**

  if $sign(W^T.tX_i) \neq y_i$ **then**
    $W \leftarrow W + y_i.X_i$
  end if
end for
return W

This algorithm converges when the input data is linearly separable as shown by F.

Rosenblatt, 1962.

The *perceptron* was quickly disregarded as unuseful when it was proven unable to solve the, now, simple problem of learning a XOR logical gate, the boolean function

$$f(x_1, x_2) = \begin{cases} 0, & x_1 = x_2 \\ 1, & x_1 \neq x_2 \end{cases}$$

.

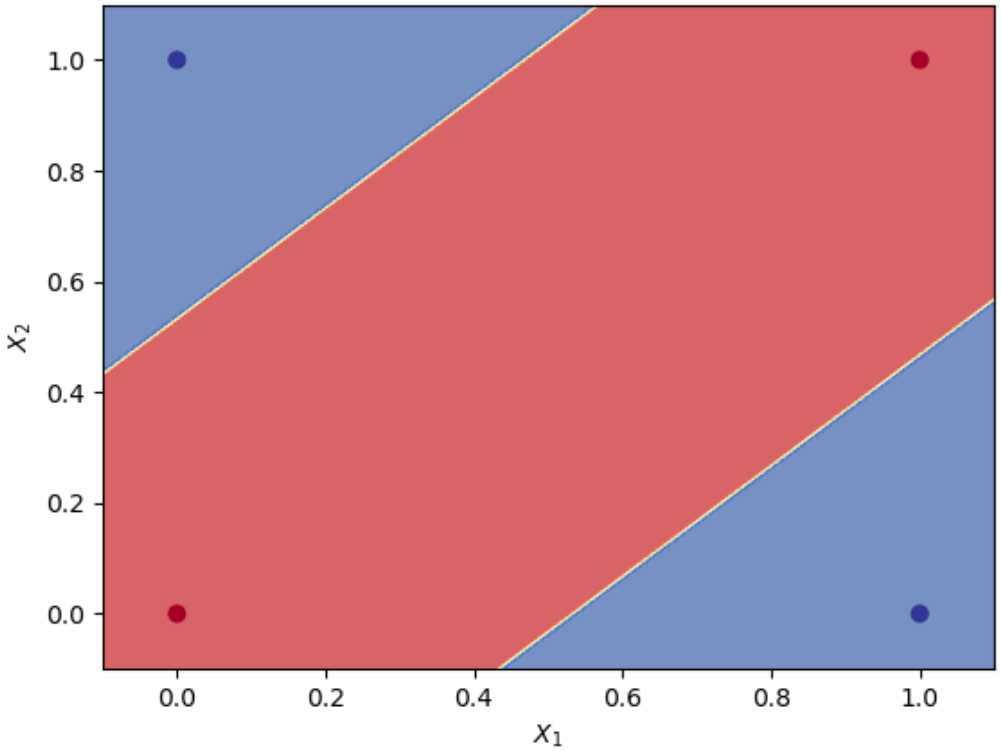

Figure 4.1: *Example of a decision boundary for the XOR problem*

That is because the perceptron is nothing but a linear regressor, while the boundary to separate data points for the XOR problem is non linear (see 4.1), something not expressable for a single perceptron.

4.2 Beyond a single Perceptron

What Neural Networks try to do is build on the idea of the *perceptron* in a way that captures the non linearity of a given function, by combining multiple *perceptrons*-like units called *neurons* into one single model. That in itself is not enough to achieve the wanted effect, after all it is just a linear model of linear models, or a superparameterized linear

model. But by introducing the concept of *hidden layers*, it is possible to escape linearity. With that, is created the fundamental architecture called Multi Layered Perceptrons (4.2), or MLPs.

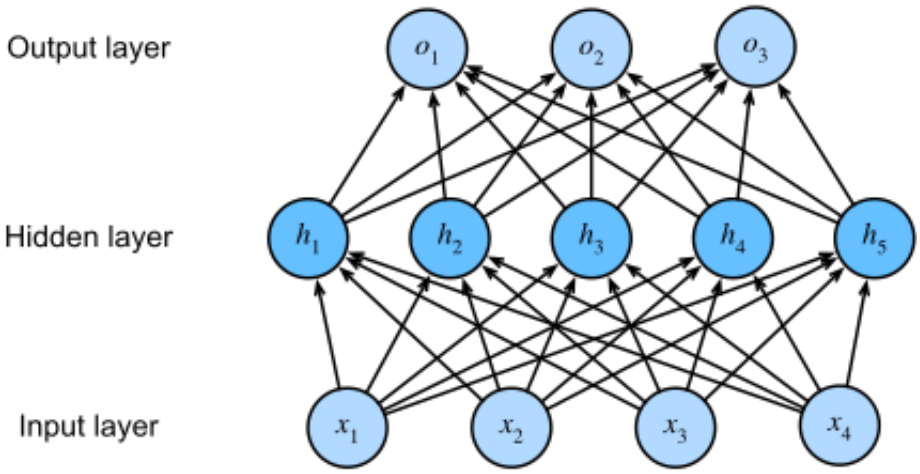

Figure 4.2: *Example of a Multi Layered Perceptron architecture, from* ZHANG *et al., 2021*

Hidden layers are intermediate layers of *neurons* between the input and the output, each one fully connected to the previous one (every input from the previous layer goes into every neuron of the current layer). A hidden layer consists of multiple *neurons*, arranged horizontally (meaning that they don't have connections between themselves), and their outputs is passed through an *activation function* σ, a non linear differentiable function.

Formally, let $f : [\mathbb{R}^d, \mathbb{R}^{d.h+h.q}] \rightarrow \mathbb{R}^q$ be a one-hidden-layer MLP Neural Network, with h hidden units. The hidden layer we will be computing a weighted sum of all inputs for each *neurons* and applying an activation function to every element of the output, in other words, let $W_1 \in \mathbb{R}^{d \times h}$ and $H \in \mathbb{R}^h$ be matrices such that

$$H = X.W_1, H \in \mathbb{R}^h$$

where H a matrix of h weighted sums of the input elements for every *neuron* in the hidden layer, X is the input, and W_1 is a weight matrix where every line represents the weights that each *neuron* applied to the input. Next, every unit in the *hidden layer* applies the same activation function σ (e.g. ReLu, Sigmoid, etc.). The expression of the result of the *hidden layer* computation can be written as

$$H = \sigma(X.W_1).$$

After that, the hidden layer outputs are passed to output nodes which can be q simple linear regressors, that will make a prediction about that input. That entails that the output of the model will be given by

$$O = H.W_2, O \in \mathbb{R}^q$$

Where $W_2 \in \mathbb{R}^{h \times q}$ is a weight matrix similar to W_1 and H is the output of our hidden layer.

And, finally, the whole network can be expressed as:

$$f(X, \theta) = \sigma(X.W_1).W_2$$

where θ is a vector of all the parameters in W_1 and W_2.

This architecture is a much more expressive model because it can utilize the hidden layer outputs, called *hidden representations*, which have a non linearity aspect embbeded in them. In fact, it has been shown by Hornik *et al.*, 1989 that Neural Networks with a finite number of neurons and as much as one hidden layer can approximate any continuous function.

Learning on this architecture is a bit more complex and it couldn't be done efficiently until Rumelhart *et al.*, 1988 developed the backpropagation algorithm. For that, a loss function must be defined, which is a function that measures how far from the desired output the model output is. Suppose you have an input data matrix $X \in \mathbb{R}^{n \times d}$, and a matrix of expected outputs for each point in X, $y \in \mathbb{R}^{n \times q}$. To evaluate a model in these points, one of the most simple loss functions would be the *mean squared error*

$$J(\theta) = \sum_{i=0}^{N} (y_i - f(X_i, \theta))^2.$$

The learning procedure for a MLP goes as follows: in the *feedforward* part we compute the value of the model for all the inputs X and calculate the loss function, after that, in the *backpropagation* part we compute the gradient of the loss function with respect to all weights θ by leveraging the chain rule and calculating the gradient backwards, one layer at a time, reusing computations already done to calculate the gradient on earlier layers via the back propagation algorithm. With these gradients calculated, the parameters are adjusted in order to minimize the loss function with a chosen optimization algorithm (Gradient Descent, LBFGS, etc).

4.3 Formally Verifying a Neural Network

As shown in the aforementioned work Preto, 2021, it is possible to use Łukasiewicz Logic to represent the function created by the neural network and formally analyze its properties. Let $f : [0, 1]^n \to [0, 1]$ be a neural network, that is exactly a rational McNaughton function, which represents a prediction model. If $f(x) \geq 0.5$, we can say that the network predicts Yes and if $f(x) < 0.5$, the network predicts No. With what was presented in the thesis is possible to verify the network's reachability and robustness if we are able to construct a representation in $Ł_\infty$-*MODSAT* $\langle \varphi, \Phi \rangle$ for f.

Chapter 5

Piece Wise Linear Approximation of Neural Networks

5.1 Different ways to tackle the problem

Two ways were noted to tackle the problem of approximating neural networks by linear piece wise functions. One of them relies on the fact that the two activation functions *ReLU* (rectified linear unit) and *TId* (truncated identity function) can be easily defined as linear piece wise functions.

$$ReLU(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases} \qquad TId(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \leq x \leq 1 \\ 1, & x > 1 \end{cases}$$

Looking at a particular family of neural networks which are composed by these functions, it's possible to take advantage of ideas discussed in Preto, 2021 in order to reduce it to a rational McNaughton function which would be a linear piece wise representation of the neural network.

Another method, which is the one actually investigated in this paper, is to construct a linear piece wise function approximation by querying the network on a few different points, and obtaining a function with k pieces that is a good approximation for the network.

5.2 Preliminaries

Linear piece wise Regression is a technique used to try and find a group of simpler (*affine*) functions that are able to approximate a continuous function, which is known to be possible due to the Stone-Weierstrass Theorem Weierstrass, 1885.

Doing this for $f : \mathbb{R} \to \mathbb{R}$ is a well studied problem. The aim is to sample an unknown

function f and create a piece wise linear function such that

$$\hat{f} = \begin{cases} \alpha_1(x - b_1) + c_1, & b_1 < x \leq b_2 \\ \alpha_2(x - b_2) + c_2, & b_2 < x \leq b_3 \\ \vdots & \\ \alpha_n(x - b_n) + c_n, & b_{n-1} < x \leq b_n \end{cases}$$

where b_i is a point called *breakpoint* and determines the start of the segment that contains the ith linear piece. This approximation can also be written as a single equation explicitly expressing the error like so:

$$\hat{f} = \alpha_1(x - b_1)I_1 + \alpha_2(x - b_2)I_2 + ... + \alpha_n(x - b_n)I_n + c + \epsilon$$

where I_i are functions such that

$$I_i = \begin{cases} 1, & \text{if } b_i < x \leq b_{i+1} \\ 0, & \text{otherwise} \end{cases}$$

and ϵ is an error term that follows a normal distribution with mean 0 and variance σ^2. The goal is to create such a function focusing on minimizing the error between $\hat{f}$ and the real function value f.

5.3 Dynamic Programming

Dynamic Programming is a technique used to solve a particular set of problems that have two characteristics: *optimal substructure* and *overlapping sub-problems*.

Optimal substrucure refers to when the solution of the problem can be obtained by combining the solution of its sub-problems. That means that we can formulate the problem with a recursive relationship where the optimal solution is an expression containing the optimal solution up to a certain point of the problem.

Overlapping sub-problems refers to the fact that during the execution of a any recursive algorithm, the amount of sub-problems solved by the algorithm will be small and instead of generating new sub-problems, these will keep reappearing as part of the solution.

That is exactly the case for segmented regression.

5.4 An algorithm for segmented regression

Even though literature has it that this problem is commonly solved with Dynamic Programming, few descriptions of it can actually be found in scientific articles. Jayadev Acharya et al. in Acharya *et al.*, 2016, for the sake of completion, describes with detail a Dynamic Programming algorithm to solve segmented regression problems before introducing a new type of algorithm for these problems. The algorithm described here is

a variation of that which tries to break segments based on an input cost C for creating segments instead of specifying how many pieces the regression should have.

For this we consider the following formulation: Let (X, y) be a dataset of points sampled from an arbitrary function f, which we want to approximate. For an estimator $\hat{f}$ of f, the square error generated by the estimator, for m input-ouput pairs $(x_i, y_i)_{i=j}^{m}$ with $j, m \in \mathbb{N}$, from the dataset is

$$\sum_{i=j}^{m} (y_i - \hat{f}(x_i))^2.$$

The goal of the algorithm is to find an estimator $\hat{f}$, which should be a linear k-piece wise function, such that it minimizes the square error between the approximation and the observed data. In other words, we want $\hat{f}$ to minimize:

$$\sum_{i=1}^{N} (y_i - \hat{f}(x_i))^2$$

which, in this case, corresponds to minimizing the squared error between a subset of input points and a linear function for k linear pieces.

Let $OPT[j]$ be the minimum possible error considering only pairs $\{(x_p, y_p)\}_{p \in J}$ from X, y in the interval of indices $J = \{1, ..., j\}$. Let, also, $err(i, j)$, where $i < j$, be the least squared error generated by a segment fitted through the points lying in the interval of indices $I = \{i, i + 1, ..., j\}$. The optimal segment error for pairs with indices in J can be formulated as the recursive relationship:

$$OPT[j] = min_{i \in I}\{err(i, j) + OPT[i - 1]\}.$$

After constructing OPT, we can backtrack and find the solution which has the minimum overall squared error and determine the linear pieces.

The pseudocode for the algorithm that constructs such table OPT is as follows:

Algorithm 2 Segmented Regression by Dynamic Programming

Input

X Data matrix of points sampled
y Data matrix of outputs of the f evaluated on X
C Cost for creating a segment

Output

Cost for creating the optimal piece wise linear function

$OPT[0] \leftarrow 0$

for $j \in \{1, ..., N\}$ **do**

  for $i \in \{1, ..., j\}$ **do**
    $err(i, j) \leftarrow$ least square error for indices in the interval $\{i, i + 1, ..., j\}$
  end for
end for

for $j \in \{1, ..., N\}$ **do**
  $OPT[j] = min_{i<j}(err(i, j) + OPT[i - 1] + C)$
end for
return $OPT[n]$

The way this algorithm works is that each interval of points is treated as a sub-problem. If we have a point X_j that is the last point of the optimal segment that starts at X_i, where $i < j$, we can compute the cost, $OPT(j)$, of fitting a segment through $\{X_i, ..., X_j\}$ if we know the cost $OPT(j - 1)$. Also, by varying the value of C, we can determine how many segments the algorithm should create because the final cost value will contain C times the number of segments created.

Chapter 6

Inherent issues with this method

Before obtaining an approximation for the function, we must address two different issue:

- the order in which the points are presented to algorithm **??** matters and it generates different pieces depending on how the data is presented;
- the function, which would be the union of the pieces generated by the algorithm, is not guaranteed to be continuous.

6.1 Order matters

Taking a closer look at the algorithm, it is imperative that exists a certain order underlying the data. Imagine, for the 1 dimensional case, we have a set of points ordered by the x-axis. Suppose there are two optimal segments that divide this dataset, S_1 which is a regression for points $\{X_1, ..., X_j\}$ and S_2 which is one for points $\{X_{j+1}, ..., X_n\}$. If we purposefully create a dataset where we show $\{X_1, ..., X_{j+1}, X_j\}$ to the algorithm, it will try to construct the optimal segment of X_j using the cost for X_{j+1} which lies further in the x-axis and that will either generate a poor regression for these points or break the segment earlier than optimal. Depending on the different ways the data is presented to the algorithm, it will generate vastly different linear pieces and we need to define ways to sort it before anything else.

For the 1 dimensional case all you need is to just order the points based on the x-axis.

For greater dimensions, we experimented with two arrangements. The first was described by Acharya *et al.*, 2016: sort the data by a determined coordinate, in this case, by the first coordinate x_1 of every point. We will call this type of ordering **X1** from now on. The other one, which we will call **C**, sorts points by the values $c = x_1 - x_2$. This was created based on prior knowledge of the function and for experimentation purposes to try and induce the algorithm to generate the pieces that one can intuitively derive from looking at figure 7.1.

6.2 Simplex division: forcing continuity

The other issue that should be adressed is that this algorithm is based on a discrete set of input points, which means that between two segments created there is an interval of points of the domain that don't have a linear function "assigned" to them. Let the function $f : [0,1]^n \to [0,1]$ be the function we are trying to approximate with our algorithm. After sampling an arbitrary number of points from this function $X = (X_1, X_1, ...)$, we order them by their first coordinate (x_1) and run the algorithm with input X and theirs respective values $f(X) = y$ to get the linear pieces that best approximate this function. Suppose that, from the output, we get two consecutive linear pieces p_i, p_k. Let p_i be the linear approximation of points in the domain interval delimited by X_{i_1} and X_{j_1}, and p_k be the same thing, but for points X_k, X_l, where $i < j < k < l$, and $i, j, k, l \in \mathbb{N}$. In this setting, there are points X_q such that $X_{k_1} > X_{q_1} > X_{j_1}$, which do not have a mapping in our approximation, creating a discontinuity. To account for these points, we will partition the unmapped interval into multiple simplices to make sure the whole domain is mapped. Let's look at an example of this in practice.

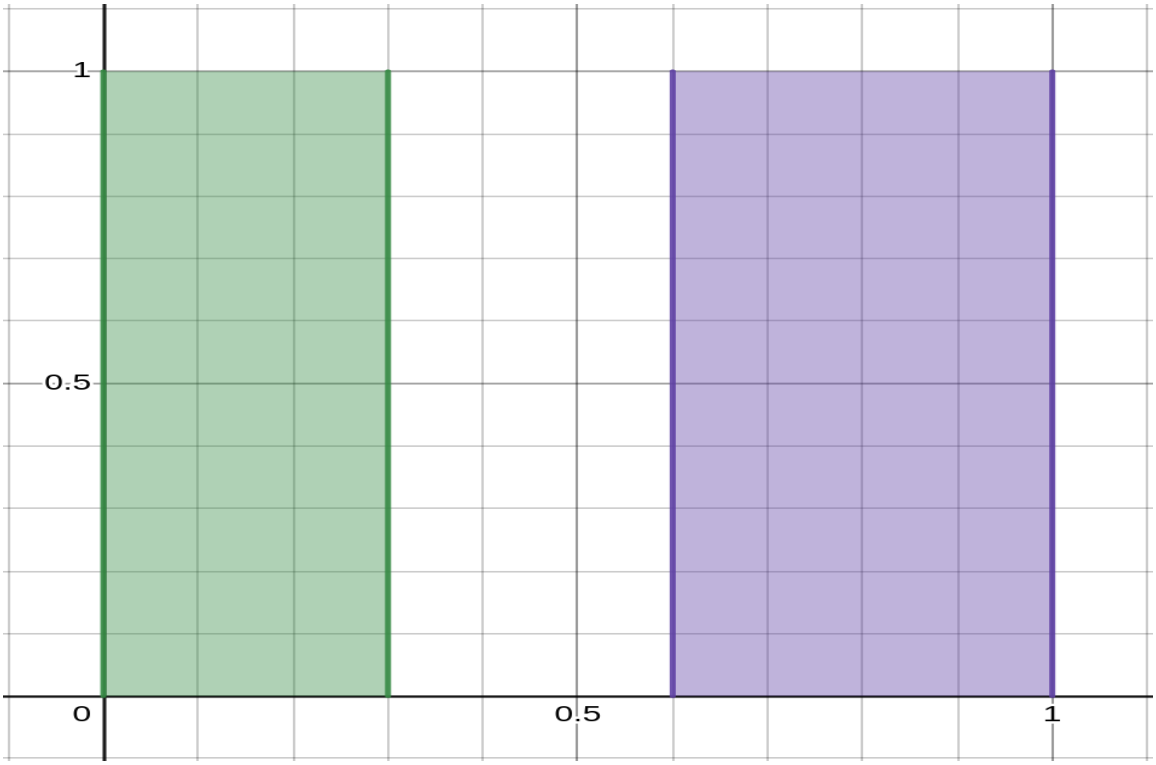

Figure 6.1: *Domain of function f partitioned by the linear pieces generated by algorithm 2*

Suppose we are trying to approximate a function $f : [0,1]^2 \to [0,1]$. If we look at figure 6.1, we can see an image of the domain of the function and the highlighted areas are the segments created by algorithm 2. We have p_1 constrained by $0 \leq x_1 \leq 0.3$ and p_2 constrained by $0.5 \leq x_1 \leq 1$. That means that our approximation as is can't produce values for points of the form $X = (x_1, x_2)$, with $0.3 < x_1 < 0.5$. To deal with this we divide the domain as illustrated by figure 6.2, with a line and create two planes delimited by the pieces and the line.

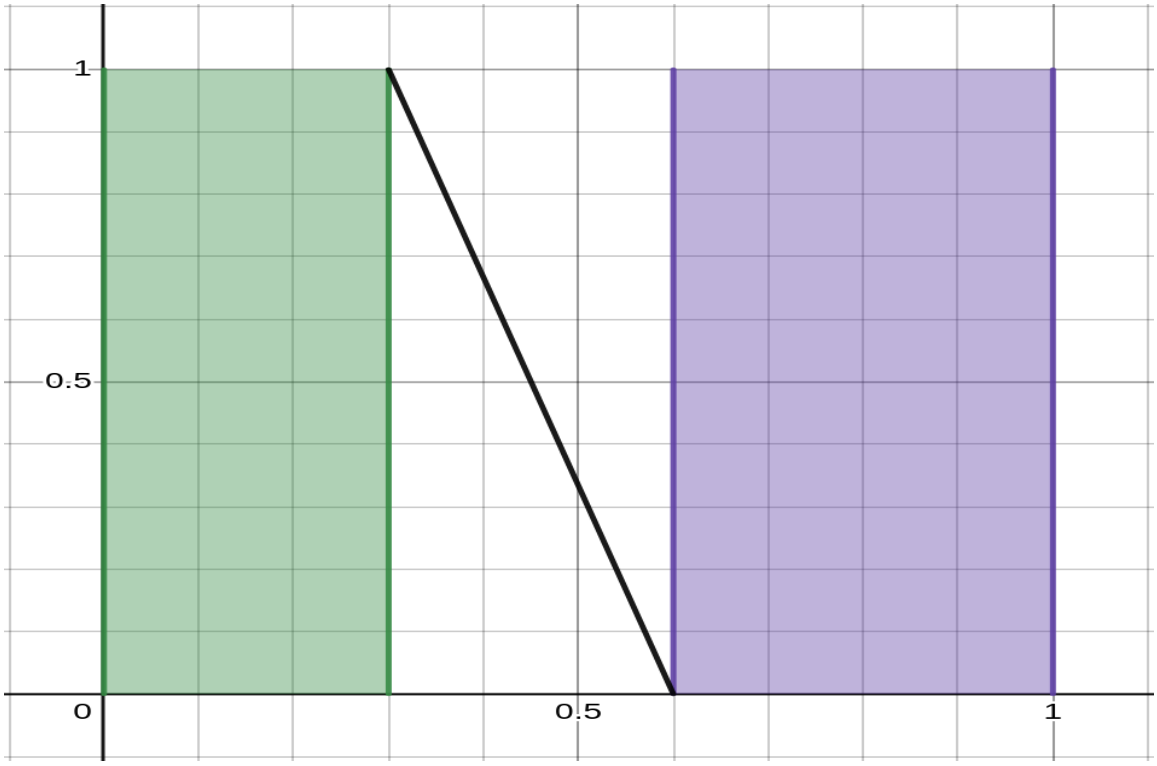

Figure 6.2: *Figure 6.1 with the delimitation of where the simplices will be created*

We can treat the problem of finding a planes in this case as determining the plane for the space $[0,1]^3$ considering points $(x, y, z) = (x_1, x_2, f(x_1, x_2))$. The equation for a plane in this case is

$$ax + by + cz + d = 0$$

where (a, b, c) are coordinates of the normal vector to that plane. To obtain a linear function from that plane we isolate the third coordinate and get

$$f(x_1, x_2) = z = (-ax - by - d)/c.$$

For determining each simplex, we would need 3 points of the form $(x_1, x_2, f(x_1, x_2))$, obtain two linearly independent vectors from them and get their cross product, the normal vector. In our example, for the first simplex, we use

$$(0, 0.3, p_1(0, 0.3));$$

$$(1, 0.3, p_1(0, 0.3));$$

$$(1, 0.5, p_2(1, 0.5))$$

and for the second simplex

$$(0, 0.3, p_1(0, 0.3));$$

$$(0, 0.5, p_2(0, 0.5));$$

$$(1, 0.5, p_2(1, 0.5)).$$

Note that, p_1 and p_2 are defined at such points and thus can be evaluated at them, giving

us all the requirements we need for partitioning. The planes, or simplices, can be seen in the domain in figure 6.3 colored red an blue, respectively. With that every point in the domain is properly mapped and we have a continuous linear piece wise approximation of our target function.

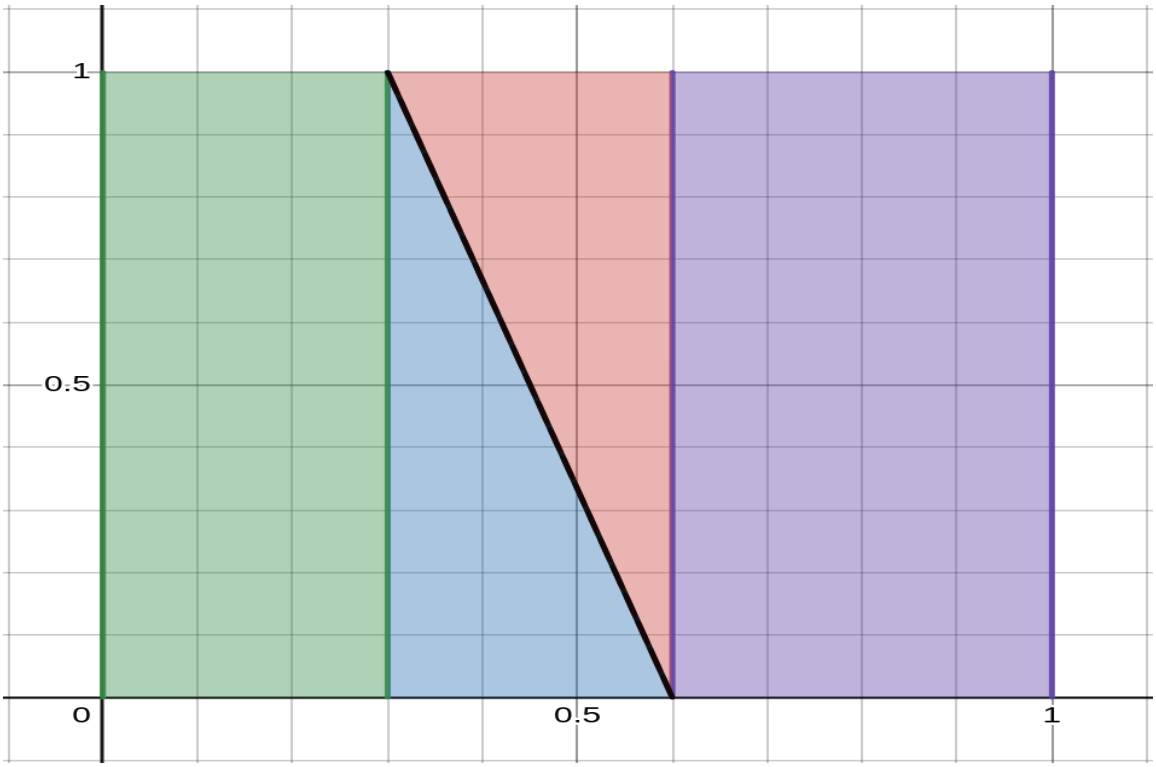

Figure 6.3: *Domain partitioned by the linear pieces generated and simplices*

Unfortunately, simplex partitioning grows exponentially in complexity as the dimension increases. In this case, we needed only two simplices to cover the unmapped region, but in $[0, 1]^7$ for example, we would need 1175 simplices at minimum Hughes and Anderson, 1996. This number grows uncontrollably, making this technique unfeasible for higher dimensions. For this reason we will be focusing on functions, or neural networks, of the form $f : [0, 1]^2 \to [0, 1]$ for this work.

Chapter 7

Approximating a trained network

7.1 Implementation

We developed an implementation for algorithm 2 (and a module to perform the discussed simplex division) in C++, leveraging the linear algebra Eigen [1], to output a continuous linear piece wise approximation of a function $f : [0,1]^2 \to [0,1]$. After that, we implemented another component to translate the output to the input of the already implement algorithm by Preto, 2021 [2] for representing linear piece wise functions with Łukasiewicz Logic.

7.2 XOR Neural Network

A simple neural network was trained in order to conduct the experiments. The network was trained to solve the XOR problem.

x_1	x_2	$x_1 \oplus x_2$
1	1	0
1	0	1
0	1	1
0	0	0

Table 7.1: *Truth table for the XOR boolean function*

The aim was for the network to learn an approximation for the function described by the table

$$f(x_1, x_2) = \begin{cases} 0, & x_1 = x_2 \\ 1, & x_1 \neq x_2 \end{cases}$$

[1] https://eigen.tuxfamily.org/

[2] https://github.com/spreto/pwl2modsat

.

The function $f : [0,1]^2 \to [0,1]$ learned by the network is presented in figure 7.1. We can look at f as a classification network such that for input $\mathbf{x} \in [0,1]^2$, if $f(\mathbf{x}) \geq 0.5$ the function outputs 1, if $f(\mathbf{x}) < 0.5$ it outputs 0.

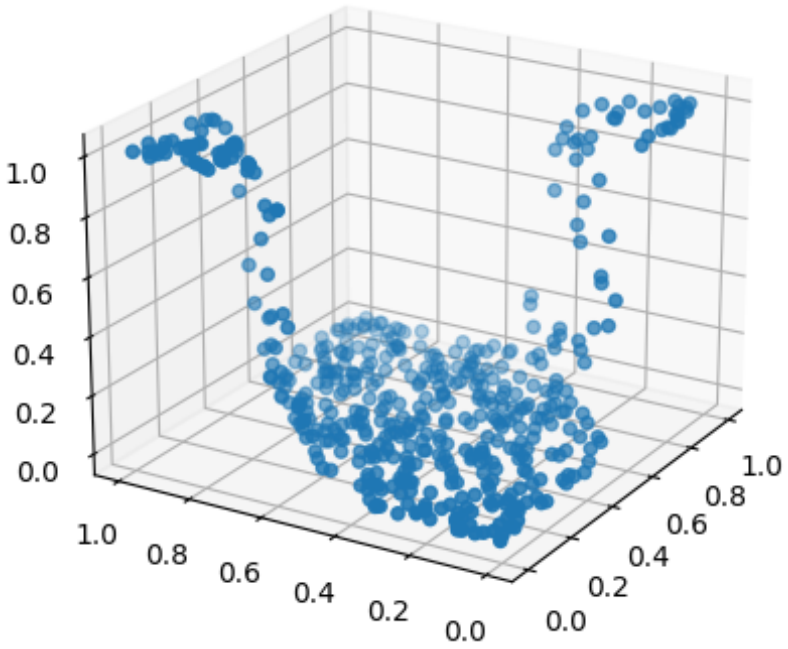

Figure 7.1: *Function learned by the neural network*

7.3 Modeling *reachability* and *robustness* in Łukasiewicz Logic

First, let us introduce the modulo satisbality representation of the constant function $\frac{1}{d}$.

$$\langle \varphi, \Phi \rangle = \left\langle Z_{\frac{1}{d}}, \{ Z_{\frac{1}{d}} \leftrightarrow \neg(d-1) Z_{\frac{1}{d}} \} \right\rangle$$

We from now on we denote by $\varphi_{\frac{1}{d}}$ the only formula in the set Φ, $Z_{\frac{1}{d}} \leftrightarrow \neg(d-1)Z_{\frac{1}{d}}$. With that, we can model the problems of *reachability* and *robustness* in Łukasiewicz Logic.

The *reachability* of a given state can be thought as the problem of determining if a neural network $f : [0,1]^n \to [0,1]$ reaches a specific probability $\pi = \frac{a}{b} \in \mathbb{Q} \cap [0,1]$.

Theorem 2. *Let $f : [0,1] \to [0,1]$ be a neural network which is a rational McNaughton function and $\langle \varphi, \Phi \rangle$ be the representation modulo satisfability of f, then f reaches probability $\pi = \frac{a}{b}$ if, and only if,*

$$\left(\bigwedge \Phi \right) \wedge \varphi_{\frac{1}{b}} \wedge a Z_{\frac{1}{d}} \leftrightarrow \varphi$$

is satisfiable.

For *robustness*, we want to verify if a neural network which is a rational McNaughton function is robust when predicting **Yes** with respect to a fixed perturbation limit $\varepsilon = \frac{\alpha}{\beta} \in \mathbb{Q}$

and to a fixed probability $\pi = \frac{a}{b} \in [0,1] \cap \mathbb{Q}$. We would like to see wether $f(x+p) \geq 0.5$, for all $x \in [0,1]^n$ and $p = (p_1, p_2, ..., p_n) \in \mathbb{R}$, such that $f(x) \geq \pi, |p_i| \leq \varepsilon$, for $i = 1, 2, ..., n$ and $x + p \in [0,1]^n$. Let $\langle \varphi, \Phi \rangle$ be the representation modulo satisfability of a neural network f which is a rational McNaughton function, for each propositional variable $X \in Var(\varphi) \cup Var(\Phi)$ we introduce X' and for each propositional variable $X_i \in \{X_1, ..., X_n\}$ we introduce a new variable P_i. Let $\langle \varphi', \Phi' \rangle$ be a representation such that all occurences of X in φ and Φ are replaced by X'.

Theorem 3. *Let $f : [0,1] \to [0,1]$ be a neural network which is a rational McNaughton function and $\langle \varphi, \Phi \rangle$ be the representation modulo satisfability of f from which $\langle \varphi', \Phi' \rangle$ is define as discussed. Then f is robust with respecto to $\varepsilon = \frac{\alpha}{\beta} \in \mathbb{Q}$ and $\pi = \frac{a}{b} \in [0,1] \cap \mathbb{Q}$ if, and only if,*

$$\begin{aligned} \Phi, \Phi', \varphi_{\frac{1}{\beta}}, \varphi_{\frac{1}{b}}, \varphi_{\frac{1}{2}}, \\ P_1 \to \alpha Z_{\frac{1}{\beta}}, ..., P_n \to \alpha Z_{\frac{1}{\beta}}, \\ (X_1' \leftrightarrow X_1 \oplus P_1) \vee (X_1' \leftrightarrow \neg(X_1 \to P_1)), \\ ..., \\ (X_n' \leftrightarrow X_n \oplus P_n) \vee (X_n' \leftrightarrow \neg(X_n \to P_n)), \\ aZ_{\frac{1}{b}} \to \varphi \vDash Z_{\frac{1}{2}} \to \varphi', \end{aligned}$$

holds.

7.4 Experiments and results

For the approximation, we sampled 250 random points from the network, order them with one of the mentioned sorting methods, and fed them to the algorithm 2. After doing the simplex segmentation of the parts of the domain that remained unmapped, we translate the output into the format taken by Preto's algorithms in order to represent the approximation in Łukasiewicz Logic using modulo satisfability. After that, we use the representation to construct satisfability and logical consequence problems in order to verify the properties we want, *reachability* and *robustness*.

The results obtained for the method **X1** of sorting the input were extremely very unsatisfatory. The algorithm generated the maximum amount of linear pieces, taking the minimum amount of neighboring points to create a plane, and doing this for all points. This yielded too many linear pieces, making it very inefficient for the SAT Solver to deal with and deeming the verification of the function untractable.

On the other hand, the pieces generated with method **C** are what one would expect to be the best pieces when looking at the graph of the function. We obtained 3 pieces generated by the algorithm plus 4 simplices used to assert the continuity of the approximation. This is much better and very feasible to solve using the Solver. In table 7.2 we present the results verified by our procedure.

Reachability		*Robustness*	
Parameters	Result	Parameters	Result
$\pi = 0.1$	✓	$\pi = 0.75, \varepsilon = 0.01$	✓
$\pi = 0.2$	✓	$\pi = 0.75, \varepsilon = 0.1$	✓
$\pi = 0.3$	✓	$\pi = 0.75, \varepsilon = 0.2$	✓
$\pi = 0.4$	✓	$\pi = 0.75, \varepsilon = 0.25$	✓
$\pi = 0.5$	✓	$\pi = 0.75, \varepsilon = 0.3$	✗
$\pi = 0.6$	✓	$\pi = 0.75, \varepsilon = 0.35$	✗
$\pi = 0.7$	✓	$\pi = 0.75, \varepsilon = 0.4$	✗
$\pi = 0.8$	✓	$\pi = 0.75, \varepsilon = 0.5$	✗
$\pi = 0.9$	✓		

Table 7.2: *Verification of an approximation of the network depicted in figure 7.1, utilizing sorting method* ***C*** *on the input points.*

As expected given the nature of the XOR problem, and corroborated by the graph of the function, the network accesses every single value between 0 and 1. As for robustness, we tried to verify that if the network is in a given state $\pi = \frac{3}{4}$ we perturb it by adding a small number to each coordinate of the input. In this particular scenario we can see that the network is fairly robust, because it continues to reach $f(x + p) \geq 0.5$ even for a perturbation of 0.25 in every coordinate of the input.

Chapter 8

Conclusions

We researched and detailed the thoeretical results used in the process of verification presented in figure 1.1. We applied such theoretical results to conduct an experiment to verify a trained neural network using Łukasiewicz Logic. During the process of implementation we encountered multiple issues, most notably the dependence of an order underlying the data and the fact that the used algorithm does not generate a continuous approximation of the function. We used a few techniques to circumvent these, but they were reliant in a few things that were available to us in this special case: prior knowledge of the function the neural network constructed and the low dimensionality of such function.

The results obtained in our experiments were in accordance to expectation and to what we knew beforehand about the function. That entails that if one is able to circumvent the issues presented, this is a valid method to create a piece wise linear approximation of a neural network.

8.1 Future Work

For future works that intend to use the aforementioned method of approximation, specially when dealing with networks in a higher dimension, one must find an efficient way to assert the continuity of the approximated function and find the optimal way of ordering the input data before executing the algorithm, if one exists.

Moreover, one can investigate the other method of representing a neural network as a linear piece wise function mentioned in chapter 5, and try to express the function as a combination of ReLU functions, which already are piece wise linear. In the case the network does not use ReLU as its actvation function, one idea is to try and create a linear approximation for the used activation (i.e. Sigmoid) and express the network as a combination of these approximated activations.

Appendix A

Proofs for Łukasiewicz Logic connectives valuations

A.1 Proof of Lemma 1

Proof.

$$\begin{aligned} v(\alpha \vee \beta) &= v((\alpha \to \beta) \to \beta) \\ v((\alpha \to \beta) \to \beta) &= min(1, 1 - v(\alpha \to \beta) + v(\beta)) \\ v((\alpha \to \beta) \to \beta) &= min(1, 1 - min(1, 1 - v(\alpha) + v(\beta)) + v(\beta)) \end{aligned}$$

If $v(\alpha) > v(\beta)$, we have $1 - v(\alpha) + v(\beta) < 1$ and:

$$\begin{aligned} v((\alpha \to \beta) \to \beta) &= min(1, 1 - min(1, 1 - v(\alpha) + v(\beta)) + v(\beta)) \\ v((\alpha \to \beta) \to \beta) &= min(1, 1 - (1 - v(\alpha) + v(\beta)) + v(\beta)) \\ v((\alpha \to \beta) \to \beta) &= 1 - (1 - v(\alpha) + v(\beta)) + v(\beta) \\ v((\alpha \to \beta) \to \beta) &= v(\alpha) \end{aligned}$$

On the other hand, if $v(\beta) > v(\alpha)$, then $1 - v(\alpha) + v(\beta) > 1$ and:

$$\begin{aligned} v((\alpha \to \beta) \to \beta) &= min(1, 1 - min(1, 1 - v(\alpha) + v(\beta)) + v(\beta)) \\ v((\alpha \to \beta) \to \beta) &= min(1, 1 - 1 + v(\beta)) \\ v((\alpha \to \beta) \to \beta) &= v(\beta) \end{aligned}$$

We can see that $(\alpha \to \beta) \to \beta$ behave exactly as the maximum function and, in fact, it is possible to define $\vee$ as the maximum of the two values. □

A.2 Proof of Lemma 2

Proof.

$$v(\alpha \wedge \beta) = v(\neg(\neg\alpha \vee \neg\beta))$$
$$v(\alpha \wedge \beta) = 1 - v(\neg\alpha \vee \neg\beta)$$
$$v(\alpha \wedge \beta) = 1 - max(v(\neg\alpha), v(\neg\beta))$$
$$v(\neg(\neg\alpha \vee \neg\beta)) = 1 - max(1 - v(\alpha), 1 - v(\beta))$$

If $v(\alpha) > v(\beta)$, we have $max(1 - v(\alpha), 1 - v(\beta)) = 1 - v(\beta)$ and:

$$v(\neg(\neg\alpha \vee \neg\beta)) = 1 - (1 - v(\beta))$$
$$v(\neg(\neg\alpha \vee \neg\beta)) = v(\beta)$$

Now, if $v(\beta) > v(\alpha)$, then $max(1 - v(\alpha), 1 - v(\beta)) = 1 - v(\alpha)$ and:

$$v(\neg(\neg\alpha \vee \neg\beta)) = 1 - (1 - v(\alpha))$$
$$v(\neg(\neg\alpha \vee \neg\beta)) = v(\alpha)$$

□

A.3 Proof of Lemma 3

Proof.

$$v(\alpha \oplus \beta) = v(\neg\alpha \rightarrow \beta)$$
$$v(\neg\alpha \rightarrow \beta) = min(1, 1 - v(\neg\alpha) + v(\beta))$$
$$v(\neg\alpha \rightarrow \beta) = min(1, 1 - (1 - v(\alpha)) + v(\beta))$$
$$v(\neg\alpha \rightarrow \beta) = min(1, v(\alpha) + v(\beta))$$

□

A.4 Proof of Lemma 4

Proof.

$$v(\neg(\neg\alpha \oplus \neg\beta)) = 1 - v(\neg\alpha \oplus \neg\beta)$$
$$v(\neg(\neg\alpha \oplus \neg\beta)) = 1 - min(1, v(\neg\alpha) + v(\neg\beta))$$
$$v(\neg(\neg\alpha \oplus \neg\beta)) = 1 - min(1, 1 - v(\alpha) + 1 - v(\beta))$$
$$v(\neg(\neg\alpha \oplus \neg\beta)) = 1 - min(1, 2 - v(\alpha) - v(\beta))$$

If $2 - v(\alpha) - v(\beta) < 1$, we have

$$\begin{aligned} &1 - v(\alpha) - v(\beta) < 0 \\ &v(\alpha) + v(\beta) - 1 > 0 \\ &max(0, v(\alpha) + v(\beta) - 1) = v(\alpha) + v(\beta) - 1 \end{aligned}$$

and also

$$\begin{aligned} v(\neg(\neg\alpha \oplus \neg\beta)) &= 1 - min(1, 2 - v(\alpha) - v(\beta)) \\ v(\neg(\neg\alpha \oplus \neg\beta)) &= 1 - (2 - v(\alpha) - v(\beta)) \\ v(\neg(\neg\alpha \oplus \neg\beta)) &= v(\alpha) + v(\beta) - 1. \end{aligned}$$

Now, if $2 - \alpha - \beta > 1$, first we have

$$\begin{aligned} &1 - v(\alpha) - v(\beta) > 0 \\ &v(\alpha) + v(\beta) - 1 < 0 \\ &v(\neg(\neg\alpha \oplus \neg\beta)) = max(0, v(\alpha) + v(\beta) - 1) \\ &max(0, v(\alpha) + v(\beta) - 1) = 0 \end{aligned}$$

and

$$\begin{aligned} v(\neg(\neg\alpha \oplus \neg\beta)) &= 1 - min(1, 2 - v(\alpha) - v(\beta)) \\ v(\neg(\neg\alpha \oplus \neg\beta)) &= 1 - 1 \\ v(\neg(\neg\alpha \oplus \neg\beta)) &= 0 \end{aligned}$$

so

$$v(\neg(\neg\alpha \oplus \neg\beta)) = max(0, v(\alpha) + v(\beta) - 1).$$

□

References

[Acharya *et al.* 2016] Jayadev Acharya, Ilias Diakonikolas, Jerry Li, and Ludwig Schmidt. *Fast Algorithms for Segmented Regression*. 2016. doi: 10.48550/ARXIV.1607.03990. url: https://arxiv.org/abs/1607.03990 (cit. on pp. 18, 21).

[S. Aguzzoli and D. Mundici 2001] S. Aguzzoli and D. Mundici. "Weierstrass approximations by lukasiewicz formulas with one quantified variable". In: *Proceedings 31st IEEE International Symposium on Multiple-Valued Logic*. 2001, pp. 361–366. doi: 10.1109/ISMVL.2001.924596 (cit. on p. 1).

[Stefano Aguzzoli and Daniele Mundici 2003] Stefano Aguzzoli and Daniele Mundici. "Weierstrass approximation theorem and Łukasiewicz formulas with one quantified variable". In: *Beyond Two: Theory and Applications of Multiple-Valued Logic*. Ed. by Melvin Fitting and Ewa Orłowska. Heidelberg: Physica-Verlag HD, 2003, pp. 315–335. isbn: 978-3-7908-1769-0. doi: 10.1007/978-3-7908-1769-0_14. url: https://doi.org/10.1007/978-3-7908-1769-0_14 (cit. on p. 1).

[Hähnle 1994] Reiner Hähnle. "Many-valued logic and mixed integer programming." In: *Ann. Math. Artif. Intell.* 12 (Dec. 1994), pp. 231–263. doi: 10.1007/BF01530787 (cit. on p. 8).

[Hornik *et al.* 1989] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. issn: 0893-6080. doi: https://doi.org/10.1016/0893-6080(89)90020-8. url: https://www.sciencedirect.com/science/article/pii/0893608089900208 (cit. on p. 15).

[Hughes and Anderson 1996] Robert B. Hughes and Michael R. Anderson. "Simplexity of the cube". In: *Discret. Math.* 158 (1996), pp. 99–150 (cit. on p. 24).

[McNaughton 1951] Robert McNaughton. "A theorem about infinite-valued sentential logic". In: *Journal of Symbolic Logic* 16.1 (1951), pp. 1–13. doi: 10.2307/2268660 (cit. on p. 6).

[Preto 2021] Sandro Márcio da Silva Preto. "Semantics modulo satisfiability with applications: function representation, probabilities and game theory". PhD dissertation. Instituto de Matemática e Estatística, University of São Paulo, São Paulo, 2021. doi: doi:10.11606/T.45.2021.tde-17062021-163257 (cit. on pp. i, iii, 15, 17, 25).

[F. Rosenblatt 1962] F. Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Cornell Aeronautical Laboratory. Report no. VG-1196-G-8. Spartan Books, 1962. URL: https://books.google.com.br/books?id=7FhRAAAAMAAJ (cit. on p. 12).

[Frank Rosenblatt 1958] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65 6 (1958), pp. 386–408 (cit. on p. 11).

[Rumelhart *et al.* 1988] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Neurocomputing: Foundations of Research*. Cambridge, MA, USA: MIT Press, 1988, pp. 696–699. ISBN: 0262010976 (cit. on p. 15).

[Weierstrass 1885] K. Weierstrass. *Uber die analytische darstellbarkeit sogenannter willkrlicher functionen einer reellen vernderlichen*. 1885 (cit. on p. 17).

[Zhang *et al.* 2021] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. "Dive into deep learning". In: *arXiv preprint arXiv:2106.11342* (2021) (cit. on p. 14).