Verification of Machine Learning

Miguel Bosch, Simone Grassi, Adrian Sondermann, Xing Su, Emese Szakály

January 2021

KEN4130: Master Research Project

Faculty of Science and Engineering
Department of Data Science and Knowledge Engineering
Maastricht University
Netherlands

Contents

1 Introduction

The purpose of this paper is to give an overview of different verification techniques used in machine learning.

1.1 Research problem

Verification of machine learning is a problem that arises from the fact that machine learning techniques provide us with solutions coming from black boxes, which translates in the uncertainty of what the system will output for unseen inputs. What we investigate in this paper is how to guarantee for a black box system, more precisely a neural network, that certain properties will hold for any given input in order to ensure that some undesired properties will never be present in the output.

More precisely, we aim to investigate the state-of-the-art to get an overview of the verification problem, and then study in depth what we consider one of the most promising approaches.

1.2 Motivation

To motivate the problem, we would like to mention a few of the several applications where verification is a favorable tool. For instance, we can use it to establish robustness guarantees against adversarial attacks, i.e. when a little targeted noise is added to an input in order to trick the model into misclassifying it. These little modifications can result in serious consequences, such as distorting a stop sign with stickers or paint could lead to an autonomous vehicle interpreting it as 'yield', causing an accident [18]. Similar example is how Tesla was fooled by stickers on the road, making the car believe that the lane is diverging and it drove into the oncoming traffic. Furthermore, verification can be applied in the field of biometric authentication, to prevent forgeries of signatures [12], fingerprints, faceprints, etc. Also, we would like to prevent situations where machine learning is used for unethical purposes, such as unfairly discriminating against ethnic minorities and women [15].

1.3 State-of-the-art

According to one categorization state-of-the-art techniques can be sound and complete or sound and incomplete. The first category has limited scalability, but they always succeed (they always give a correct answer for every question). The latter has better scalability, but it cannot answer every question (though if it gives an answer, that will be correct) [25]. One of the first published methods in this topic was from Pulina and Tacchella in 2010, called NeVeR [19], introducing an over-approximation technique, using sigmoid activation function. In 2016, Bastani et al. [4] published a techinque for evaluating a network's adversial robustness. This involves reduction from a verification-like problem to a Linear Programming problem. The LP solver is efficient and sound but incomplete,

so this means that the adversarial inputs the solver discovered are correct, but it may not found all of them. Huang et al, in 2017 proposed the DLV method [10], which applies discretization of the input space via manipulations and gives a sound and incomplete technique. In 2017 Katz et al published the sound and complete ReLuPlex method [14], which is based on the Simplex method and uses Rectified Linear Unit activation function. The AI^2 method, suggested by Gehr et al. in 2018 [8], uses over-approximation of the input property, and results again in a sound and incomplete method. Another paper in 2018, written by Ruan et al. [21], introduced the DeepGO technique which uses Lipschitz continuity and partitions the input space and bounds each piece of output with the Lipschitz constant.

1.4 Research questions

Beside the research questions raised at the beginning of our investigation of the topic, we felt the need to include one more based on our progress with the implementation of the Reluplex algorithm. The following list is the extended version of our research questions.

- What is the main motivation for developing this technology? What are the constraints and properties an algorithm would need to be a suitable solution?
- What are the main methods for solving the verification problem? Is there a method that dominates the rest? What are the strengths and weaknesses of the existing algorithms?
- Is it possible to obtain better results by combining more methods? Is it possible to improve an existing method or create a new one that behaves better in a certain environment?
- What are the possible social, legal and ethical implications of developing better verification algorithms?
- Can we implement a version of Reluplex that allows us to run experiments on it? Can we use those to derive properties of the Reluplex algorithm and observe its behaviour with diverse types of neural networks?

2 Preliminaries

In this section we will introduce the background necessary to understand the rest of the report. Since the implementation of the Reluplex algorithm [14] has been so crucial to our work, we have decided to exclude it from Subsection 2.3 which discusses relevant previous works, and instead give it a subsection on its own (2.4).

2.1 Key concepts and terminology

We now introduce the lexicon that will be present throughout the document, as well as a formalized statement of the verification problem.

- Neural network: Computing system formed by a collection of connected units called nodes, that transmit signals to other nodes, and after a number of transformations in the original input, it delivers an output. We call the set of input nodes input layer, the set of output nodes output layer and the rest of the nodes hidden layers. More information in [16].
- Activation function: A function that defines the value of a certain node of a neural network given its inputs.
- Rectified Linear Unit (ReLU): One of the simplest activation functions in which we take the max of the input and 0: $f(x) = \max(x, 0)$.
- Sound: We say that a verification algorithm is sound when, in case it provides a solution, that solution will always be correct.
- Complete: We say that a verification algorithm is complete when it always provides a valid solution.
- LP and ILP: We call Linear Programming problems (LP) to problems consisting in optimizing an objective function subject to restrictions defined by linear relationships. Integer Linear Programming (ILP) problems are very similar to LP problems, but they also allow for variables to be restricted to being integers (a non-linear restriction) [17].
- Simplex: The simplex algorithm is an algorithm used to solve LP and ILP problems. It is sound an complete [17].

2.2 Formalization of the problem statement

In this report, we refer to verification of a neural network as the problem of answering the following question:

For a neural network, an input property P, and an output property Q, does exist an input x_0 with output y_0 such that x_0 satisfies P and y_0 satisfies Q?

If such input x_0 does not exist we say that the neural network is verified for properties P and Q. It is important to note that by verified we mean that is impossible for a network with input properties P to give an output with properties Q, so Q needs to be defined accordingly.

2.3 Relevant previous work

In recent research, many verification tools have focused on using SMT (Satisfiability Modulo Theories) solvers to tackle the verification problem. SMT is the problem of deciding the satisfiability of a first order formula with respect to some theoretical formulas. It is being recognized as increasingly important

due to its applications in different communities, in particular in formal verification, program analysis and software testing. A lot of verification methods were based on SMT solving techniques, especially the Reluplex method [14], combining with linear programming, which neural networks are encoded as linear arithmetic constraints.

Furthermore, other studies have concentrated on the Symbolic Interval Analysis, which is a formal analysis method for certifying the robustness of neural networks. Any safety properties of neural networks can be presented as a bounded input range, a targeted network, and a desired output behavior. It is different from the existing solved based approaches, as it is easily parallelizable. The most famous ones are ReluVal [26] and Neurify [22], both of which are based on these interval algorithms.

2.4 Reluplex algorithm

Reluplex [14] is an algorithm for solving the verification problem based on the Simplex algorithm. It is meant to be applied only in neural networks in which the activation function is the ReLU function. To apply the Reluplex algorithm to a neural network, we define it as a set of variables and linear constraints.

In this kind of neural networks each node takes the value $v_i = \max(0, x_i)$, $x_i = \sum_{v_j \in \mathcal{L}_i} v_j w_j + b$, where w_j and b are constants and $\mathcal{L}_i$ represents the nodes in the previous layer to x_i. In the Reluplex algorithm we create two variables for each node of the hidden layers, v_i^b and v_i^f, and we keep the previous equations by letting $v_i^b = \sum_{v_j^f \in \mathcal{L}_i} v_j^f w_j + b$ and $v_i^f = \max(0, v_i^b)$.

Then we have two variables for each of the hidden layer nodes and one variable for each of the output and input layer nodes. We have the constraints taking the form defined above plus the output layer nodes constraints that have the form $v_i = \sum_{v_j^f \in \mathcal{L}_i} v_j^f w_j + b$.

The Reluplex algorithm, as the Simplex algorithm, allows for temporal violation of the constraints, so for each of the defined constraints $A_i = B_i$ we also define a new variable a_i so that $a_i = A_i - B_i$. We then call $\mathcal{X}$ the set of all the previously defined variables plus every a_i.

Then a Reluplex instance is defined by the following elements:

- **Set of basic variables** $\mathcal{B} \subseteq \mathcal{X}$: A set of variables that when initializing the algorithm contains exclusively every a_i.
- **Tableau** $\mathcal{T}$: The tableau is a matrix in which every row represents a basic variable and every column a non-basic variable. Then the i, j entry of the matrix takes value T_{ij} such that $x_i = \sum_j x_j T_{ij}$.
- **Upper and lower limits** u, l: Each variable x_i has associated an upper

and lower limit. For every v_i^b they are $[-\infty, \infty]$, for every v_i^f $[0, \infty]$ and for the input and output nodes it depends on the properties P and Q.

- **ReLU constraints** $\langle x_i, x_j \rangle$: One for each pair $(v_i^b,\ v_j^f)$. We say that a ReLU pair violates the ReLU constraint if $v_i^f \neq \max(0, v_i^b)$.

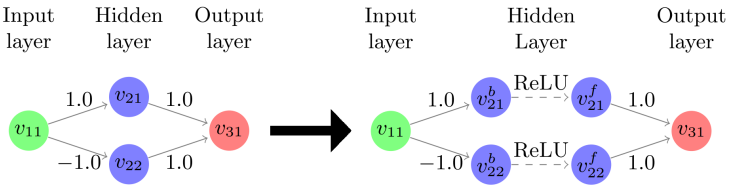

Figure 1: Example of the modelling of a simple neural network to apply the Reluplex algorithm to it, from the paper at [14]

To simplify further explanation we say $slack^+(x_i)$ if for a basic variable x_i, $\exists\, x_j \notin \mathcal{B}$ such that $T_{ij} > 0,\ x_j < u(x_j)$ or $T_{ij} < 0,\ x_j > l(x_j)$. $slack^-(x_i)$ gets defined in a similar way, with x_i such that $T_{ij} < 0,\ x_j < u(x_j)$ or $T_{ij} > 0,\ x_j > l(x_j)$.

We will say we are pivoting a non-basic variable x_j when we make it basic by swapping it with some basic variable x_i, and then updating the tableau so that the definition above still holds.

Finally we say that we split the problem at $\langle x_i, x_j \rangle$ when we make two copies of the original problem, $subinstance_1$ and $subinstance_2$. One subinstance assumes that the neuron related to the ReLU pair $\langle x_i, x_j \rangle$ is active, so $x_i > 0, x_j > 0, x_i = x_j$, and the other assumes that is inactive, so $x_i \leq 0, x_j = 0$. Then the Reluplex algorithm works as follows:

Algorithm 1 Reluplex

Input: Weights of the layers of the neural network, input and output properties P and Q and threshold t

Output: $True$ or $False$

$k(\langle x_i, x_j \rangle) \leftarrow 0 \ \forall ReLU\_Pair \ \langle x_i, x_j \rangle$
procedure RELUPLEX
  while $True$ **do**
    while $x_i \notin [l(x_i), u(x_i)]$ for some $x_i \in \mathcal{X}$ **do**
      if $x_i \in \mathcal{B}$ **then**
        if $x_i < l(x_i)$ **and** $slack^+(x_i)$ **then**
          Update the corresponding x_j to increase x_i
        else if $x_i > u(x_i)$ **and** $slack^-(x_i)$ **then**
          Update the corresponding x_j to decrease x_i
        else
          return $False$
      else
        Pivot x_i
    while Some $ReLU\_Pair \ \langle x_i, x_j \rangle$ violates ReLU constraints **do**
      if $k(\langle x_i, x_j \rangle) > t$ **then**
        Split at $\langle x_i, x_j \rangle \rightarrow subinstance_1, subinstance_2$
        if $subinstance_1 = True$ **or** $subinstance_2 = True$ **then**
          return $True$
        else
          return $False$
      else if $x_k \in \mathcal{B}$ **and** $x_l \notin \mathcal{B}$ for $k, l \in (i, j)$ **then**
        Update x_l so the ReLU constraint is satisfied
        $k(\langle x_k, x_l \rangle) := k(\langle x_k, x_l \rangle) + 1$
      else if $x_i \in \mathcal{B}$ **then**
        Pivot some x_k so $x_i \notin \mathcal{B}$
      else
        Pivot x_i

The Reluplex algorithm is both sound and complete, and since theoretically it might need to split the problem one time for each neuron, its complexity is 2^n times the complexity of solving the corresponding LP problem, although usually it behaves better than that.

Its soundness and completeness comes from the fact that it is based on Simplex, which is known to be also sound and complete, and in worst case scenario more information can be found at [14].

Reluplex: Example

For a better understanding we would like to introduce how Reluplex works in practice on a small neural network. Let us consider the one in Figure 2 and test

if it is satisfiable that the input is in the range $[0,\ 1]$ and the output falls into the interval $[6,\ 8]$.

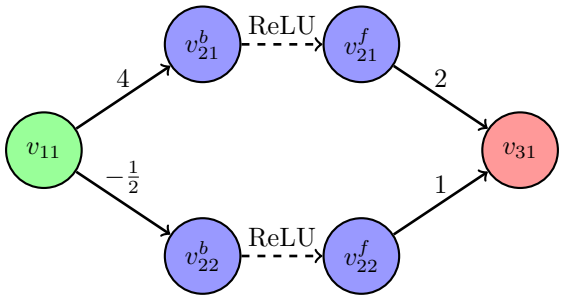

Figure 2: Neural network with splitted ReLU nodes

Introducing the new basic variables $a_1,\ a_2,\ a_3$, the initial tableau is made up from the following equations:

$$a_1 = -4v_{11} + v_{21}^b \quad (1)$$

$$a_2 = \frac{1}{2}v_{11} + v_{22}^b \quad (2)$$

$$a_3 = -2v_{21}^f - v_{22}^f + v_{31} \quad (3)$$

Initially we assign value 0 to every variable. Also, we also need to set lower and upper bounds to each of them. For the input and output variable we set their bounds accordingly to the desired property we want to check. The variables $a_1,\ a_2,\ a_3$ are forced to be 0, i.e. both their lower and upper bounds are set to 0, in order to ensure that the equations (1), (2) and (3) hold. The hidden nodes are split into backward and forward facing ReLU nodes. The forward facing ones need to be nonnegative, but the backward facing ones have no restriction. According to these considerations we can set up the initial configuration as Table 1 shows.

variable	v_{11}	v_{21}^b	v_{21}^f	v_{22}^b	v_{22}^f	v_{31}	a_1	a_2	a_3
lower bound	0	$-\infty$	0	$-\infty$	0	6	0	0	0
assignment	0	0	0	0	0	0	0	0	0
upper bound	1	∞	∞	∞	∞	8	0	0	0

Table 1: Initial configuration

The starting basic variables are $a_1,\ a_2,\ a_3$ and the ReLU pairs are $< v_{21}^b, v_{21}^f >$ and $< v_{22}^b, v_{22}^f >$.

To begin with, we check if all the variables are within their bounds. v_{31} violates its bounds, therefore we need to update it. Since it is a non basic variable, we can simply assign its new value to it so that it reaches its lower bound:

$\alpha(v_{31}) = 6$. But now a_3 got updated to 6, as well, and therefore it violates its bounds. Being a basic variable, it cannot just simply be updated, but need to be switched with a non basic variable first. For this reason we pivot a_3 with v_{21}^f, and then update it back to 0. As a result, the equation (3) will be changed to

$$v_{21}^f = -\frac{1}{2}a_3 - \frac{1}{2}v_{22}^f + \frac{1}{2}v_{31}. \tag{4}$$

Based on equation (4) the new assignment for v_{21}^f is 3. Now all variables are within their bounds, but the ReLU constraint is violated at the pair $< v_{21}^b, v_{21}^f >$. The variable v_{21}^b is non basic, therefore we can update it easily, assigning the new value to it: $\alpha(v_{21}^b) = 3$. However, this changes the value of a_1, as well, and because it is basic, we have to pivot it first with v_{11}, and then we can update it back to 0. The new equation resulting from this is the following:

$$v_{11} = -\frac{1}{4}a_1 + \frac{1}{4}v_{21}^b. \tag{5}$$

The current tableau is made up from the equations (5), (2) and (4) and the new value assigned to v_{11} is $\frac{3}{4}$. Now v_{11} is a basic variable, thus we have to substitute it with the right hand side of equation (5) whenever it appears on the right hand side of other equations (with the non basic variables). As a result, equation (2) needs to be rewritten as follows:

$$a_2 = -\frac{1}{8}a_1 + \frac{1}{8}v_{21}^b + v_{22}^b. \tag{6}$$

At the moment the basic variable a_2 has the assignment $\alpha(a_2) = \frac{3}{8}$. In order to be able to update it back to 0, we need to pivot it first with the non basic variable v_{22}^b. This brings us to a new form of the equation (6):

$$v_{22}^b = a_2 + \frac{1}{8}a_1 - \frac{1}{8}v_{21}^b. \tag{7}$$

After updating a_2 to 0, the value assigned to v_{22}^b, based on equation (7), will be $\alpha(v_{22}^b) = -\frac{3}{8}$.

Currently all variables are within their bounds, and all ReLU constraints hold. As a consequence, the problem turned out to be SATISFIABLE. The final tableau is made up from the following equations.

$$\begin{aligned} v_{11} &= -\frac{1}{4}a_1 + \frac{1}{4}v_{21}^b \\ v_{22}^b &= a_2 + \frac{1}{8}a_1 - \frac{1}{8}v_{21}^b \\ v_{21}^f &= -\frac{1}{2}a_3 - \frac{1}{2}v_{22}^f + \frac{1}{2}v_{31} \end{aligned}$$

Below Table 2 shows with which values the algorithm has stopped.

variable	v_{11}	v_{21}^b	v_{21}^f	v_{22}^b	v_{22}^f	v_{31}	a_1	a_2	a_3
lower bound	0	$-\infty$	0	$-\infty$	0	6	0	0	0
assignment	$\frac{3}{4}$	3	3	$-\frac{3}{8}$	0	6	0	0	0
upper bound	1	∞	∞	∞	∞	8	0	0	0

Table 2: Final configuration

2.5 Other algorithms

Next to Reluplex, there are several different approaches to the Machine Learning verification problem, which have been tested and that we investigated during this project. In the following, we list some state-of-the-art algorithms we selected and tested to gain insight into how they work, which are their flaws and strengths and possibly come up with further improvements.

2.5.1 MIPVerify

Tjeng et al. [24] formulated verification problems of piecewise-linear neural networks as a mixed integer program (MIP). Therefore, MIPVerify assesses the bounds on the nodes to tighten the constraints, before it solves an adversarial problem that tries to estimate the maximum allowable disturbance on the input side. This method is both sound an complete.

2.5.2 Neurify

Neurify [22] is an efficient formal verification system for analyzing self-defined properties on given neural networks. It leverages symbolic linear relaxations based on symbolic interval analysis to provide tight output approximations. For cases unproved, it can further use linear solver to cut down false positives. In general, it is able to scale to large networks (e.g., over 10,000 ReLUs).

2.5.3 Planet

Planet is a verification approach presented by Ehlers [6], which generates a linear approximation of the overall network behavior that can be added to SMT or ILP instances which encode verification problems. In the end, Planet operates by explicitly attempting to find an assignment to the node phases. Thus, it relies on optimization similar to MIPVerify [24].

However, it does not include any heuristic as Reluplex does to determine which decision variables should be split over and it is incomplete but sound [6].

2.6 Verification frameworks

Over the last years many different verification algorithms have been introduced. However, they use inconsistent input and output formats and offer varying support for (deep) neural network architectures. Thus, comparing verifiers empiri-

cally as well as choosing the best one per task becomes more and more difficult. In order to address this issue, the use of a framework for running network verifiers with a common input/output format is the consequent solution.

Two existing frameworks can be found within the Marabou Framework developed by Katz et al. [13] or the Deep Neural Network Verification (DNNV) Toolbox [23].

3 Methodology

The focus of our project has been explanatory. We have picked known and well developed solutions to the verification problem and we have tried to deepen the understanding of the characteristics of those methods, and the state-of-the-art of the field, and then we have tried to reflect on the consequences of that.

To do this we divided our focus on two main branches: investigating Reluplex, which we considered it was a nice and mature approach to the verification problem after our preliminary research; and investigating the state-of-the-art, comparing all the alternatives we could find for the verification problem.

To investigate Reluplex, we decided to implement our own version of the algorithm, which is described below, and we have run very different tests that allow us to further investigate the properties of the algorithm. We also have speculated with the possibilities of the Reluplex algorithm and present an alternative that might be worth investigating in the future.

The theoretical comparison investigation has been mostly relevant to get a clear understanding of the state-of-the-art and being able to select the algorithm we wanted to focus on. In the early stage of implementation, we compared different methods and finally chose the Reluplex as our baseline algorithm.

Then we present an analysis of the implication of both branches of work, plus our reflection on the state of the verification problem, the impact it has right now in the field of machine learning and the relevance it might have in the future.

3.1 Reluplex adaptations

To better meet the requirements of our project, we have performed some variations to the original Reluplex implementation, that can be found at [20].

First and foremost we have modified the way the program gets its inputs. The implementation linked to the original paper was made to check 10 predetermined properties of a certain set of neural networks called ACAS Xu networks, which are used as airborne collision avoidance systems for unmanned aircraft [11].

However, we wanted to be able to work with neural networks of diverse size and deepness, to evaluate the behaviour of the Reluplex algorithm at different tasks,

as well as being able to input any desired property for those neural networks, instead on some properties exclusive to ACAS Xu netowrks. That is why we decided to implement our version with more flexibility at the inputs, allowing it to verify any desired property of any neural network.

We also made some slight modifications in the algorithm described at [14], which make no difference on its theoretical properties but that allowed us to implement it more easily. An example of this can be found in our function `pivot` that is defined to have the functionality of the functions **Pivot$_1$, Pivot$_2$** and **PivotForRelu** of the original paper.

Finally, we face some problems regarding the computation time: first of all our algorithm is not optimized towards running time and it is not implemented in a fast programming language like C++, and we also do not have access to the same computation resources that the authors of Reluplex had. This is why our algorithm is bound to be much slower and this is why we have added a timeout in the form of a counter to our implementation, so we can safely run experiments with the limits we have chosen.

4 Implementation

One of the main results of the project is to provide with our own implementation of the Reluplex algorithm. To do this we have followed the structure of the algorithm detailed in Section 2, with some slight changes. An overview of the implementation can be found at Figure 3 and we describe the implementation details below.

Through the implementation, we say that a problem is SATISFIABLE (SAT) when the neural network is verified, so the output cannot have property Q as long as the input has property Q. When this is not the case, and we can find a counterexample, we say that the problem is UNSATISFIABLE (UNSAT).

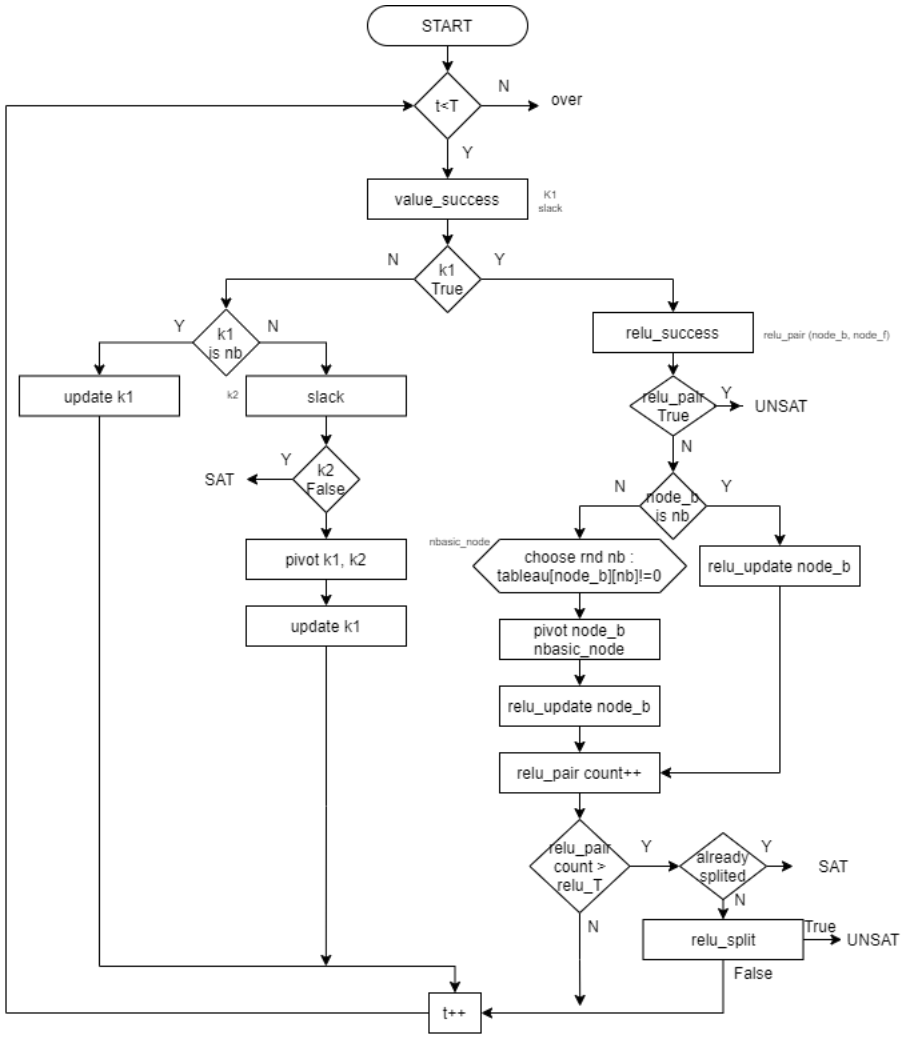

Figure 3: Flow chart of our implementation of the Reluplex algorithm

4.1 Data structures

To rightfully choose the data structures for the implementation we had to consider two requirements: Reluplex needs a lot of iterations to run, and in every iteration it accesses the data structures. For this reason the interaction complexity has a strong impact on the performance; on the other side, to make the results, the code and the debug more understandable, we added some information about the network's architecture in the variables.

The algorithm is contained within a class **`Reluplex`**. To initialize it it needs as an input a matrix **`layers`** that contains the weights of the neural net-

work we want to analyze. To set the properties P and Q the function **`set_bound(node, bounds)`** must be called.

The main structure is a dictionary called **`variables`**, which contains a variable for each $x \in \mathcal{X}$. To guarantee an access with complexity $O(1)$, it contains an array with the following information about the variables:

- **`key`**: An integer value that represents the variable. It is unique for each variable.
- **`value`** α: A float that represents the value of the variable.
- **`tableau position`**: An array with a **label** that represents the type of the variable, 'n' for non-basic and 'b' basic (which correspond to the tableau columns or rows); and an integer being the **index** of the tableau row/column where the variable is currently located.
- **`neural network position`**: An array to link the variable and the node that it represents. It consist of: the layer (from input 1 to output n), the number of the node in that layer (starting at node 0) and the type ('b' for the variables generated by the constraints in the simplex algorithm, 'f' for v_i^f nodes and 'n' for the rest).

For example an entry with the form of **`key : [3.0, ['n', 2], [1, 0, 'n'] ]`** means that the variable **`variables[key]`** is representing the node number 0 of the input layer, which at this moment is located in the 3$^{\text{rd}}$ column of the tableau, and has value 3.0.

The rest of the elements that the algorithm requires are compiled in the following structures:

- **`tableau`**: A matrix that keeps the information of the tableau $\mathcal{T}$.
- **`basic_variables`** and **`nonbasic_variables`**: Two arrays of keys that keep track of the order of the variables in **`tableau`**.
- **`value_constraints`**: A dictionary that maps **`key`** to a tuple of values representing the lower and upper bounds l, u of a variable.
- **`relu_constraints`**: An array containing tuples of **`key`** corresponding to the variables that have ReLU relationships.
- **`relu_count`**: An array that keeps track of the number of times a ReLU pair has been updated.

4.2 Pseudocode

Algorithm 2 Reluplex Implementation

Input: `layers`, bounds of inputs and outputs by calling `set_bounds(node, bounds)`

Output: `True` (SAT) or `False` (UNSAT)

function CREATE_TABLEAU(`layers`)
    Initialize `variables` by creating all variables corresponding to a_i and the nodes
    Initialize `b_var` as all variables corresponding to a_i and `nb_var` as the rest of variables
    Initialize `tableau`
    Initialize `value_constraints`
function PIVOT(`b, nb`)
    `c` = `tableau[b, nb]`
    `tableau[b, j]` = $-$`tableau[b, j]`/`c` $\forall$`j` $\neq$ `nb`
    `tableau[b, nb]` = $1/c$
    `tableau[i, j]` = `tableau[i, nb]` $\cdot$ `tableau[b, j]` + `tableau[i, j]` $\forall$`i` $\neq$ `b`, `j` $\neq$ `nb`
    `tableau[i, nb]` = `tableau[i, nb]` $\cdot$ `tableau[b, nb]` $\forall$`i` $\neq$ `b`
    Update `b_var` and `nb_var`
function SLACK(`b, pos`)
    for `key, value, j:=tableau_pos` in `variables` **do**
        `l, u`:=`value_constraints[key]`
        if `pos` is `True` **then**
            if `tableau[b, j]` $>$ 0 and `value` $<$ `u` **then**
                return `variables[key]`
            if `tableau[b, j]` $<$ 0 and `value` $>$ `l` **then**
                return `variables[key]`
        else
            if `tableau[b, j]` $<$ 0 and `value` $<$ `u` **then**
                return `variables[key]`
            if `tableau[b, j]` $>$ 0 and `value` $>$ `l` **then**
                return `variables[key]`
    return `False`
function VALUE_SUCCESS
    if `val` in `value_constraints[key]` $\forall$`val, key` in `variables` **then**
        return `key, True`
    return `True`
function RELU_SUCCESS
    if not `node_f` $=$ max(0,`node_b`) $\forall$`node_b, node_f` in `relu_pair` **then**
        return `node_b, node_f`
    return `True`

```
function UPDATE(nb)
    Update value of variable variables[nb] to have value in code-value_constraints[value]
    delta = value - old_value, j:=variables[nb].tableau_position
    value_b = value_b + delta · tableau[b, j] ∀b in b_var, value_b in variables[b].value
function RELU_UPDATE(node_b, node_f)
    Update value of variable variables[node_b] to be equal to that of node_f
    delta = value - old_value, j:=variables[node_b].tableau_position
    value_b = value_b + delta · tableau[b, j] ∀b in b_var, value_b in variables[b].value
function RELU_SPLIT(node_b, node_f)
    reluplex_1:=reluplex with value_constraints[node_b] = [0, inf]
    reluplex_2:=reluplex with value_constraints[node_f] = [0, 0], value_constraints[node_b] = [-inf, inf]
    return reluplex_1 or reluplex_2
function RUN(T, relu_T)
    for i=0; i ¡ T; i++ do
        k1, slack_dir = value_success
        if k1 is True then
            relu_pair = relu_success
            if relu_pair is True then
                return True
            node_b, node_f = relu_pair
            if node_b in nb_var then
                relu_update(node_b, node_f)
                relu_count(relu_pair) := relu_count(relu_pair) + 1
            else
                pivot(any basic var b, node_b)
                relu_update(node_b, node_f)
                relu_count(relu_pair) := relu_count(relu_pair) + 1
            if relu_count(relu_pair) > relu_T then
                return relu_split(node_b, node_f)
        else
            if node_b in nb_var then
                update(k1)
            else
                k2:= slack(k1, slack_dir)
                if k2 is False then
                    return False
                else
                    pivot(k1, k2)
                    update(k1)
```

4.3 Implementation details

In order to avoid problems with excessive execution times, our version of the Reluplex algorithm implements a threshold T passed as parameter. When a max number of iterations is surpassed, the algorithm terminates with the corresponding state.

The algorithm starts by checking if all variables have their value within their bounds. If that is the case, the function **`value_success()`** returns **`True`**, otherwise it returns the variable that needs an update and its slack. The slack represents the update direction needed, **`True`** means that the variable has to increase, False that it has to decrease.

Depending on the return value of the function value_success(), **`k1`**, the algorithm will perform two very different actions in that current iteration: this first, corresponding to the left-side branch of the flow chart at Figure 3, concerns the update of the values to fit the value constraints; the second (rith-side branch) is about the violation of the ReLU relations.

Regarding the first case, we can directly update **`k1`** if it is a non-basic variable, whereas if it is basic the algorithm needs to pivot before the updating. To do this, it is necessary to properly choose a non-basic variable to switch with **`k1`**. By properly we mean that the variable we choose will not go out of bounds once we update **`k1`** to be within its own bounds. That decision is made by the function **`slack()`**. If it returns **`False`**, there are no more variables that could be switched and the problem is SAT, otherwise the return represents the variable named **`k2`**. Then we can safely pivot **`k1`** with **`k2`** and then update **`k1`**, since it will now be a non-basic variable.

When **`k1`** is **`True`** (all the variables are within their bounds) the algorithm checks that the ReLU relations are satisfied, using the function **`relu_success()`**. This function could return **`True`**, and this means that the problem is solved with status UNSAT, or a duple of variables linked by a ReLU relation that is not satisfied.

Due to our algorithm's structure and the neural network architectures, it simplifies the implementation to update the variable corresponding to v_i^b, so the ReLU pair satisfies the constraint (we will refer to that variable as **`node_b`**, and the variable corresponding to v_i^f as **`node_f`**). As before, if that variable is non-basic it is possible to update it directly by **`relu_update()`**, while if it is basic, we need to pivot.

The ReLU updating phase might lead to infinite looping, which is dealt with with splitting. However splitting is computationally intensive, so to avoid it as much as possible we introduce randomness in the basic variable we pivot to update the variable **`node_b`**. This way we can try different possibilities that might allow us to avoid splitting. However there will be cases in which splitting is unavoidable, and we will perform it whenever the number of updates of a ReLU pair grows over a predefined threshold.

Due to how our functions are implemented, the function **`relu_split()`** might chose to split a ReLU pair that is already split. If that is the case, then we consider that the problem is SAT. Otherwise the function **`relu_split()`** divides the problem in two sub-problems: the first one with **`node_b`** activated, which means that **`node_b`** has to be greater or equal than 0, and **`node_f`** has to be equal to **`node_b`**; the second sub-problem with **`node_b`** inactive, so **`node_b`** is free to assume every negative number and **`node_f`** is always set to zero. If one of the suproblems returns UNSAT, the original problem is UNSAT too, otherwise it is SAT.

5 Experiments, results and analysis

In this section we present the experiments we have done with our implementation of the Reluplex algorithm, and the reasoning behind them. We also present a case study that serves as an example for the usage of the algorithm.

5.1 Results and analysis

To see the scalability of our algorithm to random problems, we ran several tests. The neural networks we use as given system, have different number of layers (3,4,5,6,10); always one input and one output node, and varying number of hidden nodes. We made two different set of experiments with distinct strategies on how to choose the weights of the network and which input-output properties to check. The details of these experiments and their results can be found in the following two subsections.

5.1.1 Experiments with random weights

During this set of experiments we work with random weights and the property we test is whether it is satisfiable that the input is in the interval $[0, 1]$ and the output is within 0.5 and 1. In the meanwhile we count the number of different operations needed before the algorithm stops, namely the number of iterations, updates, pivots and splits, in order to see the tendency they follow as we change the architecture of the neural network.

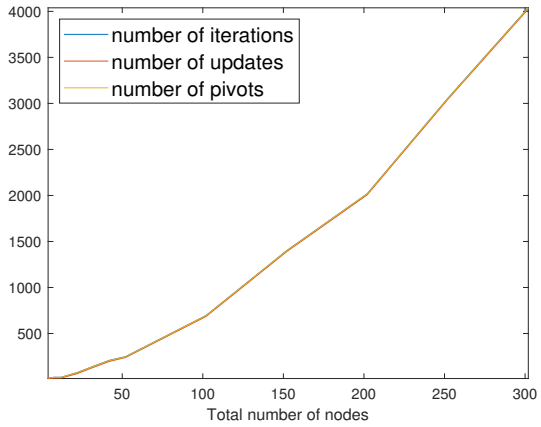

Figure 4: The number of iterations, updates and pivots done as the total number of nodes increases in a 3-layer neural network.

In order to get these results, we ran four tests for each particular neural network structure, and averaged the values we gained from these. What we experienced, is that within the tests belonging to one architecture, the variance is low, therefore this amount of test seems sufficient for reliable results within this framework.

First, we fixed the number of layers to be 3, and increased the number of hidden nodes through 12 steps in the network from 2 to 300, to see the trend of how the count of the functions used by Reluplex changes. The results are illustrated in Figure 4, and we can realize that the numbers are going up remarkably, supporting the fact that the complexity of the algorithm grows quickly, making computations by hand extremely difficult. In fact, as we raise the number of hidden nodes from 2 to 300, the number of iterations executed by the program goes up from 20 to slightly over 4000 quite fast. As we see on Figure 4, the three curves move nearly equally together, they have only a very slight difference in their values, which cannot really be detected at this order of magnitude from the plot.

Then we ran tests for higher number of hidden layers, as well, to see how much this property influences the number of times each function has to be executed during the algorithm. In order for this to be comparable with the previous test results, we designed the structure of the networks so as each of them has the same number of total nodes spread through the hidden layers.

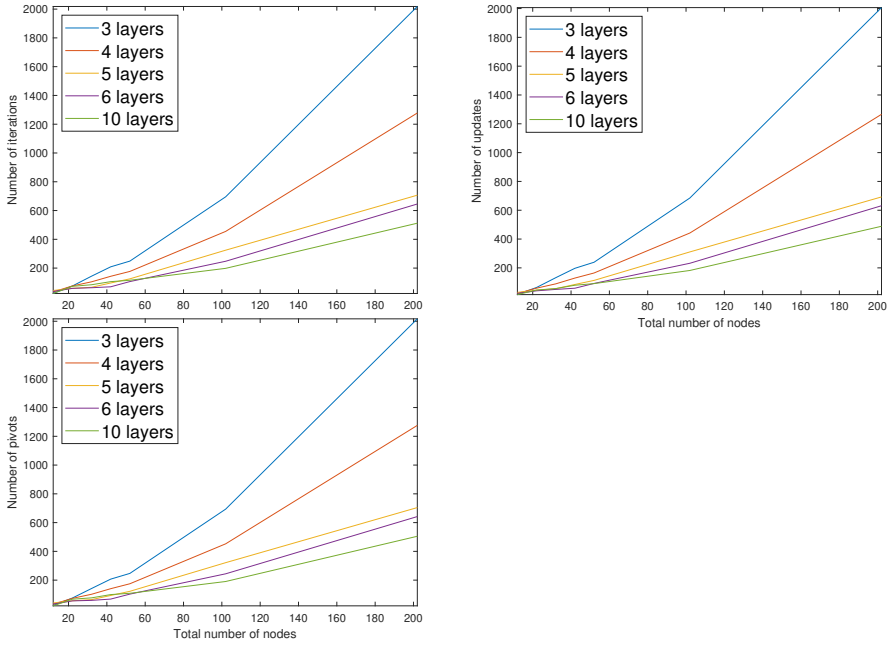

Figure 5: The count of iterations, updates and pivots in function of the increasing number of nodes for different number of layers.

As Figure 5 shows, the number of iterations, updates and pivots follow the same trend, with really similar numbers. From the graphs we see that the curve belonging to the 3 layers has the sharpest incline, and in this example as the number of layers grow, the curves start to flatten a bit. However, even for the 10 layers, which has the smallest numbers, when going from 2 to 200 hidden nodes, it has a 25 times higher value for each of these functions.

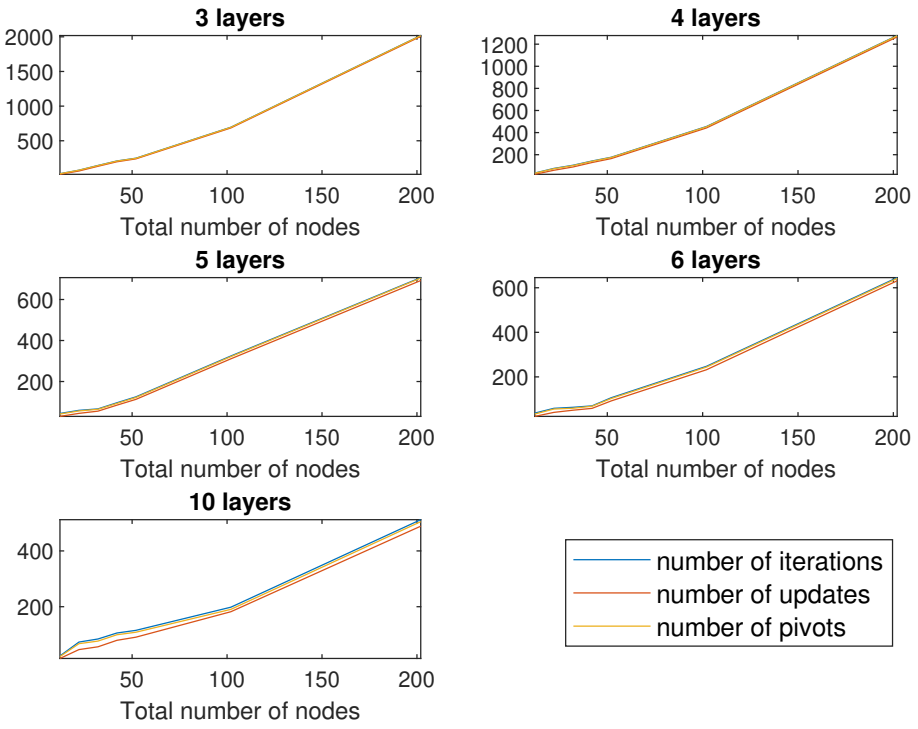

Figure 6: Comparision of the count of different functions when running Reluplex on neural networks with different number of layers.

It is noticeable from Figure 6 that the tendencies of the observed functions are very alike for each particular size of network, they only differ in the degree of increase.

So as to compare the number of splits done for each structure, we collected not only the averages, but also the highest number of splits done, because in itself the average for this function is not representative enough as these numbers are quite small and similar for all structures. As we can observe it from Figure 7 and 8, the highest number of splits occur at the networks with 10 layers, and the lowest at 3 layers. We can also notice that except for the structures with 6 and 10 layers, the values almost stagnate, as the size of the network grows.

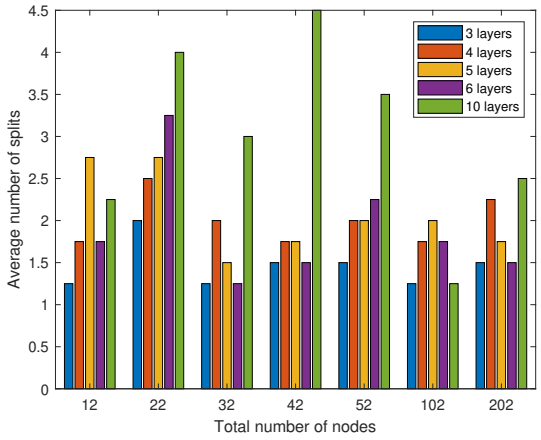

Figure 7: Comparision of the average number of splits needed when running Reluplex on neural networks with different number of layers.

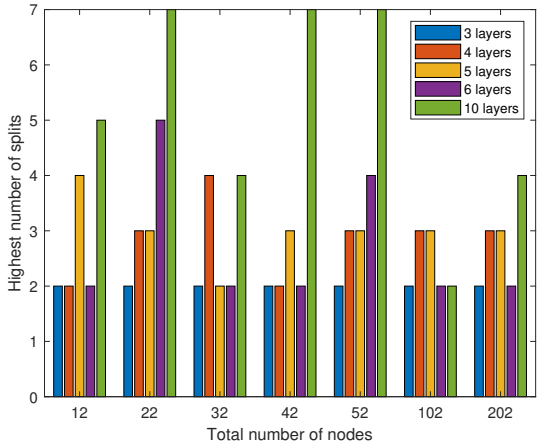

Figure 8: Comparision of the highest number of splits needed when running Reluplex on neural networks with different number of layers.

5.1.2 Experiments with pseudo-random weights

In this case we test for neural networks that do not have completely random weights, but instead some weights that are the result of some training. To do this we have created a random dataset for training consisting on random numbers in the interval $[0, 1]$ for the inputs x and for the outputs we have used the function $y = f(x) = x + 1 + \epsilon$, where ϵ is independently distributed at random from a normal distribution $N(0, 0.1)$.

Then we have performed the exact same tests as before, in order to being able to

compare the results, but with the property $Q = [3, \infty]$ instead. This experiment is designed to test the performance of Reluplex in an environment in which the weights could actually be the result of training a neural network. Even when the task at hand is so simple, we can see that it can resemble more practical cases in which we aim to balance an output around a certain value, without letting it go over a certain threshold.

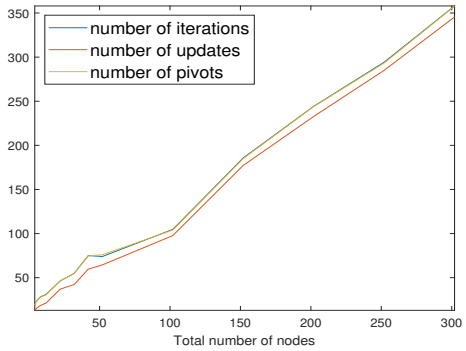

Figure 9: The number of iterations, updates and pivots done as the total number of nodes increases in a 3-layer neural network.

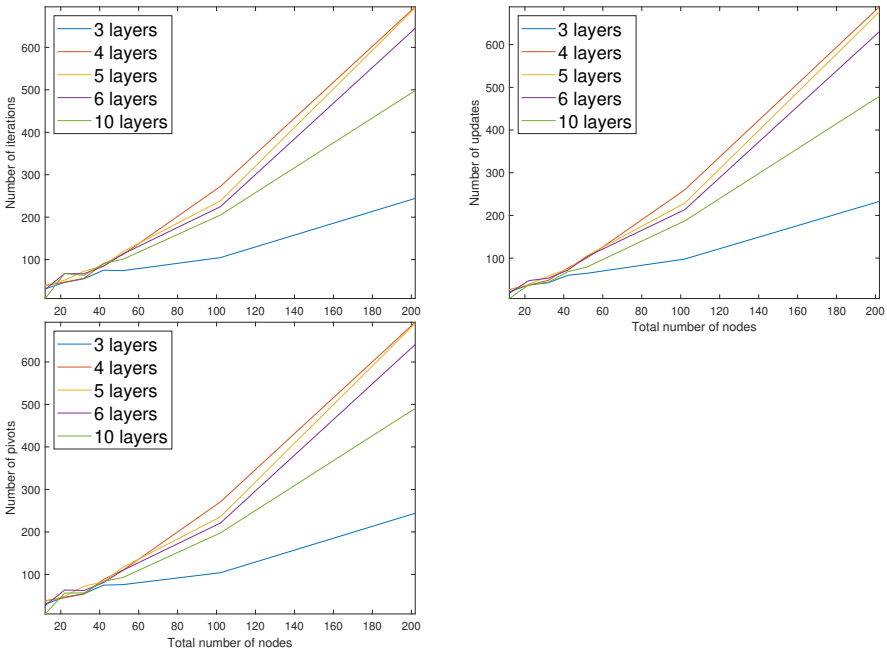

Figure 10: The count of iterations, updates and pivots in function of the increasing number of nodes for different number of layers.

We can see at Figure 9 the results for the 3-layer experiment and at Figure 10 those for different number of layers.

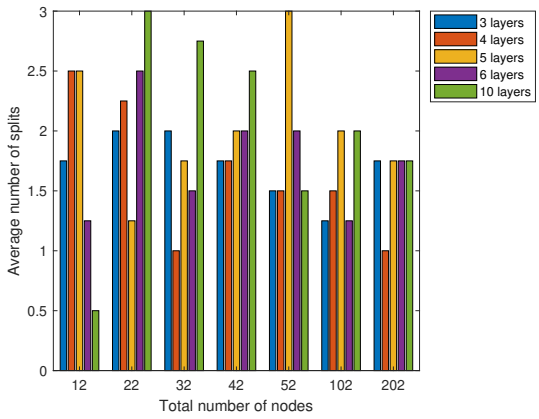

Figure 11: Comparision of the average number of splits needed when running Reluplex on neural networks with different number of layers.

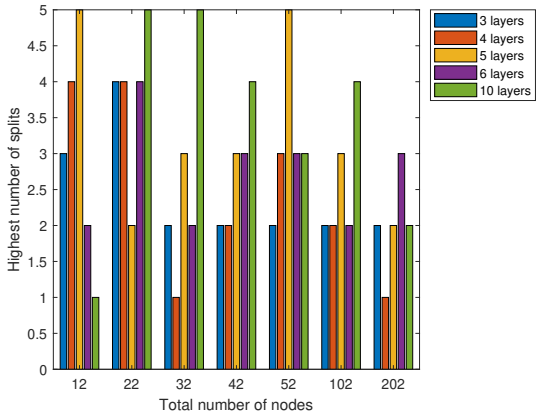

Figure 12: Comparision of the highest number of splits needed when running Reluplex on neural networks with different number of layers.

5.2 Case study

In previous sections we have been analysing the Reluplex algorithm from an abstract perspective. Now we will present a case study to provide of a realistic example of the usage of the Reluplex.

We will use a simple neural network to control an automatic brake system of a car which has two sensors: one for detecting the vehicle speed s and one for measuring its distance d from an obstacle, e.g. another car that has suddenly stopped. These sensors are appropriately conditioned, by an external circuit, to be in the range [0, 1]. The output of the neural network is a magnitude b, that represents the torque that needs to be applied to the brakes in order to avoid collision, as it is shown in Figure 13.

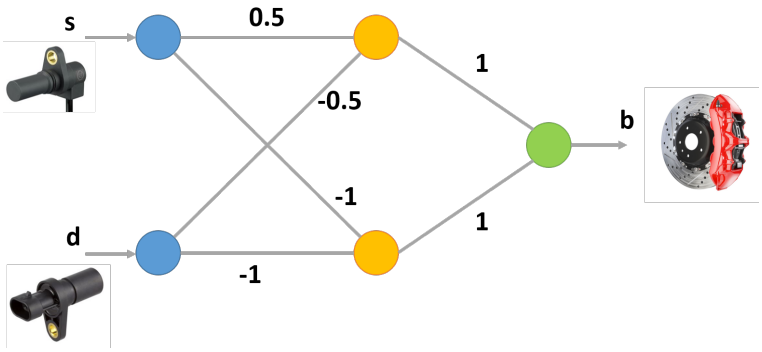

Figure 13: The representation of our brakes control system, with two sensors as input and an electric actuator as output

To be efficient, a brake needs to avoid the blocking and consequent slippage of the wheels, since that situation would increase the space required for successfully braking and can even be dangerous. In order to guarantee this behaviour we impose a constraint: for every distance, if the speed of the car is too high (greater or equal than 0.5) the torque applied to the brakes has to be lower or equal than 0.5.

$$\forall d \in [0,1], \forall s \in [0.5,1] \Rightarrow b \in [0,0.5]$$

A brute force approach could be to discretize the state space and test every possible input combination. Since we have chosen a particularly simple neural network with only two input this could be doable and we could even represent the corresponding function as a 3D graph (Figure 14).

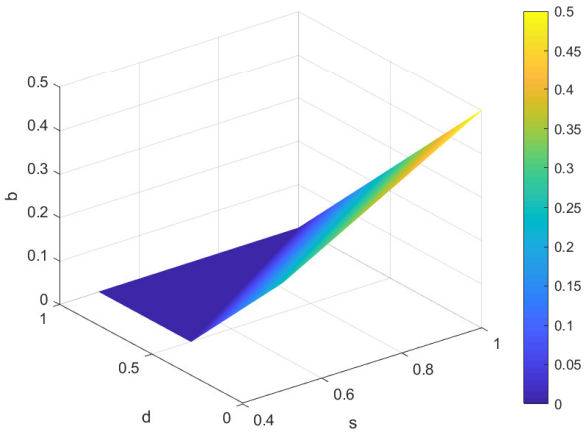

Figure 14: Representation of the breaking process based on equation (8)

$$b = \max(0.5s - 0.5d, 0) + \max(-s - d, 0) \tag{8}$$

We see that the neural network behaves as we would like to: when the distance d decreases, we start to brake at a lower speed, and the braking torque is directly proportional to the speed itself. If the distance is higher than 0.5, we don't need to activate the breaking system, and in none of the situations there is a blocking of the wheels. The problem of this approach is that it is unfeasible for most neural networks, since the computational complexity escalates very quickly and because we are discretizing the state space we cannot guarantee that the results hold for every possible state space.

We can instead use the Reluplex algorithm to search for a counterexample. We give as inputs the weight matrices of the neural network, and set the bounds of the input nodes s and d at [0.5, 1] and [0, 1], and the output node b at [0.5, 1], respectively. Then we search for a state (a set of value for each nodes) that makes for a feasible solution of the Reluplex algorithm, making it return **True**, meaning that the problem is UNSATISFABLE.

In our case study the outcome of the algorithm is SATISFABLE, as we could already see in the surface plot. It is important to note that this case study is a simplification of a much more complex system that would be an automatic braking system, but it serves as an example of the power of the Reluplex algorithm and the superiority of using validation methods compared to brute force approaches, specially in more complex networks where the latter is unfeasible.

6 Discussion

We have evaluated the results we have gotten and used them to induce the behaviour of the Reluplex algorithm in different scenarios. We also present potential improvements for the algorithm and our ethical, social and legal overview of the verification of neural networks problem.

6.1 Algorithm properties

To get an insight on the properties of the Reluplex algorithm we have performed the two types of experiments mentioned above, random weights and pseudo-random weights.

First of all, we need to note that when we are testing in random weights neural networks, the probability of a network with random property P satisfying a random property Q is almost zero. This makes for all the random tests to return `False` (SAT). This limits the insight we can get about the whole behaviour of the algorithm with this type of experiments, but allows us to get a lot of information about Reluplex in instances that are satisfiable.

Regarding the number of iterations in a 3-layer network, we can see at Figure 4 that it increases approximately linearly with the number of nodes. This tendency repeats itself with any number of layers (Figure 6). We can see that in instances that are trivially satisfiable (we call trivially satisfiable the instances in which the probability of it being not satisfiable is very low) the algorithm has a linear complexity.

We also see at Figure 6 that there is some influence of the number of layers on the complexity, the lower the number of layers the more iterations we need for the algorithm to output a result. However, the influence of the number of layers in the complexity is not too great, since the number of equations from which the Reluplex variables and tableau are derived from, depends only on the number of nodes, and not on the architecture of the network.

At Figure 5 it is seen that the function executions are really similar with one another, excluding the splits, and the complexity of those functions is also very similar, so there is not a clear bottleneck for the execution of the algorithm. For the splits, at Figures 7 and 8 we can see that the number of splits is not very high in any case, especially considering that its theoretical bound is 2^n.

When testing the pseudo-random weights networks, the first thing we can observe is that the number of iterations decreases considerably. In general we can say that the algorithm behaves better. This is true not only for the number of iterations but for the number of splits too, as seen in Figures 11 and 12.

Interestingly enough, we can see that the influence the number of layers had in the complexity no longer has a linear effect (the more layers the less iterations needed) but instead we find that the 3-layered networks need the less iterations, and the rest seem to keep a similar relationship as before (Figure 10).

Based on the results we can say that the differences between using Reluplex in completely random networks as when using them in more realistic ones is relevant. Although since we change the thresholds the experiments are not exactly the same, there is a strong indication that it behaves better in the realistic networks.

6.2 Potential improvements

When working with the Reluplex algorithm, a question naturally arises: can we adapt the Reluplex algorithm to work with other activation functions? If we could that would be a great improvement on the usefulness of the algorithm.

However, this is not a simple task, because the simplicity of the ReLU activation function is what allows the algorithm to be complete. More precisely, it is the fact that every node can only have two states, active or inactive, that allows for a splitting in a binary tree of the solutions.

In fact, the Reluplex algorithm's soundness and completeness are based on the soundness and completeness of the Simplex algorithm, and the fact that, in the worst case, we calculate a Simplex instance 2^n times, thanks to the previously explained capacity to split (for the detailed proof check [14]).

Nevertheless, as we can see at the experiments at Section 5, the number of splits that the algorithm does is usually low, so we could take advantage of that and try to adapt the algorithm to work with other activation functions. This is possible since the only function that requires the ReLU properties is the split function, so if we tried to make the algorithm work with another activation function, say the logit function $f(z) = \log \frac{z}{1-z}$, we could do it by performing some modifications in the algorithm.

The main modification would be that although division of the neurons into v_i^b and v_i^f would still be done, the node v_i^f would change to represent $v_i^f = \log \frac{v_i^b}{1-v_i^b}$ instead. Then the function checking for the success of the ReLU function **ReluSuccess**, and the function for updating pairs **ReluUpdate** would need to adapt to meet the new requirement.

This modifications would mean that the algorithm would no longer be complete, since we cannot split the conflicting pairs. However, because it is still based on the Simplex algorithm, it would still be sound, and it would be able to solve some neural networks.

We could even go further and try to apply some splitting rules, that still would not guarantee the completeness, but could allow it to solve more complex networks. E.g. splitting by adding constraints in the conflicting node, so in one instance $v_i^b \in [0, 0.5], v_i^f \in [-\infty, 0]$ and the other one $v_i^b \in [0.5, 1], v_i^f \in [0, \infty]$

6.3 Comparison

A lot of verification methods are based on SMT solving techniques, specifically the Reluplex method. The key insight is to avoid testing paths that mathematically can never occur, which allows testing neural networks that are orders of magnitude larger than was previously possible, for example, a fully connected neural network with 8-layers and 300 nodes each.

Symbolic interval analysis is another formal analysis method for certifying the robustness and verifiable of neural networks. Any safety properties of neural networks can be presented as a bounded input range, a targeted network, and a desired output behavior. ReluVal [26] and Neurify [22] is based on these interval algorithms, and it is different from the existing solved based approaches. In their paper, on the ACAS Xu [11], Neurify [22] as the update method of the ReluVal [26], the performance is 5000 times better than the Reluplex.

Naturally, there still are a lot of verification tools, but when we are trying compare them, the problem is there are inconsistent and non-standard input and output formats. In order to solve such problems, a large number of frameworks have been developed nowadays, such as the Marabou [13] and DNNV [23] we introduced in Subsection 2.6. Marabou is derived from the development of Reluplex, which is an SMT-based tool that can answer queries about network attributes by converting queries into constraint satisfaction questions. Besides, it can accommodate networks with different activation functions and topologies and can perform advanced reasoning on the network, which can reduce the search space and improve performance. Another algorithm is DNNV [23], and its big advantage over Marabou is that it supports a large number of different algorithms such as Planet [6], BaB [5], Neurify [22], etc.

Unfortunately, all these verification frameworks implementations are done for specific networks and properties, and thus compatibility with new algorithms is quite low. This also means that the comparison between those algorithms is a hard task, since they are designed to meet very different requirements.

Compatible expansion of supporting algorithms of these verification frameworks may be one of the future research directions.

6.4 Ethical, social and legal overview

Nowadays artificial intelligence is taking hold in every area of our lives, we have bots, vocal assistants, self-driving vehicles, recommender systems, and health assistants.

A considerable amount of those systems are based on neural networks. These systems showed to be very reliable and make our life easier, however, the obvious consequence is that we accept more and more on their decisions.

For example, when we are driving, usually we follow the indication of our navigator, and if it proposes a completely new path, we believe it; thinking that

probably there is an accident or cars stuck on the old way and our navigator knows it better. The natural evolution of this behaviour will be an overall system that decides autonomously how to address the cars, choosing for us.

However, there are still issues to be solved in those kind of systems. One of the main issue is the one we address on this project, that we do not have a completely clear understanding of these systems, we agree on their power but we usually use them as a black box: giving an input and trusting the outcome, without caring about the reasoning behind it. We are setting the control of our lives in systems that we do not understand [9].

Another important fact in the neural networks' nature is that they need human knowledge, in the form of a dataset, for learning how to solve a problem. Our knowledge is full of biases, they came from our societies' structure, from the way we have learned the concepts, from the way we have used to get the data. Moreover, despite the fact that there are a lot of techniques to get a dataset that is Identically and Independently Distributed (IID) and as unbiased as possible, we cannot avoid the creation of unprepared reasoning path in our networks.

For instance, the military trained a classifier to distinguish enemy tanks from friendly ones, to use it on weapons. It resulted to perform very well on the dataset, but poorly on the field, and it was later discovered that the cause was that the network learned to recognize the weather in the photos and not the tanks [1].

To be in advance on the technology race, in 2018 the European Union introduced the General Data Protection Regulation (GDPR), a series of data protection and privacy principles and regulations to protect the personal data of EU citizens and residents. Despite some weakness showed from scholars such as Maastricht University's Maja Brkan [2], it is undeniable that it is a great step towards regularization of the artificial intelligence uses.

The GDPR's article 13 comma 2 let. f [7] states that the controller has to provide "meaningful information about the logic involved, as well as the significance and the envisaged consequences of such processing for the data subject", but this is not always possible because a black box system does not provide any information about its reasoning.

Unfortunately, to completely explain why a neural network makes a decision is an incredibly complex task. To do this, the state-of-the-art is moving into the direction of the architecture visualization, inspired by the revolution that the discovery of the microscope brought to the biology understanding. The problem is that explanation and accuracy are usually in zero sum games, to increase one means decreasing the other. Thus, the importance of verification in the sense that it could be enough to guarantee reliability and impartiality.

Checking that our network fits certain constraints allows us to prevent unexpected and potentially dangerous behaviours.

E.g, in the 1970s and 1980s St. George Hospital Medical School in London used

a computer program to screen the applicants that was discriminating against woman and ethic minorities [15]. COMPAS score (used to predict the risk of crime recidivism) has strong ethnic biases [3], and the last years saw the boost of the predictive justice based on artificial intelligence, hence it is clear how dangerous unverified neural networks can be.

A verification system could bind the outcome in an acceptable variation when the input feature is completely free to assume every possible value.

In conclusion of this high level overview, it is evident that using a powerful instrument without understanding it could bring very hard problems in the future. These systems will control more and more parts of our lives and their intrinsic ethic and reasoning will be under the control of the manufactures. In a not so far future, Mercedes will have the power to choose if a the driver of a child on the pedestrian crossing will live when an incident will happen.

The law has already started to move in the right direction to prevent and show a guideline to the programmers, showing the importance of investigating this field, to keep on track with legal and moral standards.

7 Conclusion

As we expressed in our problem statement 1.1, the purpose of this paper was first of all to give an overview about the state-of-the-art verification techniques that exist in the field of machine learning and then go into details of these methods as much as possible. During our work, we managed to investigate different techniques for the verification problem and also, we implemented our own version of Reluplex, one of the most promising algorithms in this field. Moreover, the latter allowed us to run several experiments, and therefore we were able to conclude some particular properties about this algorithm. In the next paragraphs we address all our previously stated research questions and aim to answer them according to our best knowledge gained through the process of our research.

In both the discussion and the introduction we have showed several cases that motivate the need of further investigation in the field of verification, such as the examples in [18] or [15]. The ability to have security guarantees in some applications of neural networks is enough motivation to invest in developing verification algorithms.

To answer the question "What are the possible social, legal and ethical implications of developing better verification algorithms?" we presented an essay with our educated opinion in the matter in Subsection 6.4.

Within the state-of-the-art, we found that there is no one method that dominates the others. Each algorithm has its advantages and disadvantages. Generally speaking, based on the test of Reluplex in Acas xu [11] as a benchmark, the methods based on symbolic interval, like Neurify and ReluVal, run fast as

they are and can easily be parallelized. On the other hand, methods based on Mix integer programming perform better regarding robustness and verifiability. In fact, all those methods require specific input and property formats, which is why it is hard to give a ranking.

Even though the field is not mature, in Section 5 we have shown that the Reluplex algorithm can be used in a wide variety of different neural networks, and thus it could be applied to real life problems. Although our implementation is not powerful enough to run on very complex networks yet, the work at [14] has shown that it is also feasible to use it in deep neural networks.

With reference to the research question we added later during the project, seeing our progress, we also performed some experiments that allowed us to derive some interesting properties of the Reluplex algorithm. From the random experiments we can deduce that it performs almost linearly with satisfiable networks, and that it seems not to be influenced much by the architecture of the network. These properties make the algorithm versatile in the sense that it can be used in any type of network, and useful, since its speed for satisfiable networks allows us to either verify a network or raise a suspicion that it might be non-verifiable. We can also see that those properties hold for more realistic neural networks, with the pseudo-random experiments, although the generality of those properties in more complex networks has yet to be proven.

Regarding our initial research question of exploring the possible improvements of the existing methods, we can conclude that due to the complexity of the task it is very difficult to achieve a remarkable improvement on the scope of this project. However, we propose an adaptation of the Reluplex algorithm to other activation functions that might have some potential to be explored.

In conclusion, the verification problem is very complex, but we consider it to be a necessity in the future, and therefore it is worth studying. We believe that the Reluplex algorithm has great potential as a base of this field, thanks to its properties, and we expect some other algorithms to be developed based on it in the future.

References

[1] A.A.Freitas. "Comprehensible classification models: a position paper". In: *SIGKDD Explorations Newsletter* 15 (2014), pp. 1–10. DOI: 10.1145/2594473.2594475.

[2] James Alford et al. *GDPR and Artificial Intelligence*. 2020. URL: www.theregreview.org/2020/05/09/saturday-seminar-gdpr-artificial-intelligence.

[3] Julia Angwin et al. *Machine Bias*. May 2016. URL: www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing.

[4] Osbert Bastani et al. "Measuring Neural Net Robustness with Constraints". In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. NIPS'16. Curran Associates Inc., 2016, pp. 2621–2629. ISBN: 9781510838819.

[5] Rudy Bunel et al. "Piecewise Linear Neural Networks Verification: A comparative study". In: *arxiv:1711.00455* (2017).

[6] Ruediger Ehlers. "Formal verification of piece-wise linear feed-forward neural networks". In: *International Symposium on Automated Technology for Verification and Analysis*. Springer. 2017, pp. 269–286. DOI: 10.1007/978-3-319-68167-2_19.

[7] *GDPR (General Data Protection Regulation full) english*. URL: www.gdpr-info.eu.

[8] Timon Gehr et al. "AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation". In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2018, pp. 3–18. DOI: 10.1109/SP.2018.00058.

[9] Y.N. Harari. "Homo Deus, Breve storia del futuro". In: *Homo Deus, Breve storia del futuro*. Bompiani, 2018. ISBN: 9788845298752.

[10] Xiaowei Huang et al. "Safety Verification of Deep Neural Networks". In: *Computer Aided Verification* (2017), pp. 3–29. DOI: 10.1007/978-3-319-63387-9_1.

[11] K. Julian et al. "Policy Compression for Aircraft Collision Avoidance Systems". In: *IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*. 2016, pp. 1–10. DOI: 10.1109/DASC.2016.7778091.

[12] A. Julita et al. "Online Signature Verification system". In: *5th International Colloquium on Signal Processing & Its Applications*. 2009, pp. 8–12. DOI: 10.1109/CSPA.2009.5069177.

[13] Guy Katz et al. “The Marabou Framework for Verification and Analysis of Deep Neural Networks”. In: *Computer Aided Verification*. Springer, Cham, July 2019, pp. 443–452. DOI: 10.1007/978-3-030-25540-4_26.

[14] Guy Katz et al. “Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks”. In: *Computer Aided Verification* (2017), pp. 97–117. DOI: 10.1007/978-3-319-63387-9_5.

[15] Stella Lowry and Gordon Macpherson. “A blot on the profession”. In: *British Medical Journal* 296.6623 (1988), pp. 657–658. DOI: 10.1136/bmj.296.6623.657.

[16] Warren McCulloch and Walter Pitts. “A Logical Calculus of Ideas Immanent in Nervous Activity”. In: *Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133. DOI: 10.1007/BF02478259.

[17] Katta G. Murty. *Linear Programming*. John Wiley & Sons, 1983. ISBN: 9780471097259.

[18] Nicolas Papernot et al. “Practical Black-Box Attacks against Deep Learning Systems using Adversarial Examples”. In: *ArXiv* abs/1602.02697 (2016).

[19] Luca Pulina and Armando Tacchella. “An Abstraction-Refinement Approach to Verification of Artificial Neural Networks”. In: *Computer Aided Verification*. Springer-Verlag, 2010, pp. 243–257. DOI: 10.1007/978-3-642-14295-6_24.

[20] *Reluplex algorithm implementation*. https://github.com/guykatzz/ReluplexCav2017. Accessed: 2020-11-13.

[21] Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. “Reachability Analysis of Deep Neural Networks with Provable Guarantees”. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*. July 2018, pp. 2651–2659. DOI: 10.24963/ijcai.2018/368.

[22] Wang Shiqi et al. “Efficient Formal Safety Analysis of Neural Networks”. In: *32nd Conference on Neural Information Processing Systems (NIPS)*. Montreal, Canada, 2018.

[23] David Shriver. *Deep Neural Network Verification Toolbox*. 2019. URL: https://github.com/dlshriver/DNNV.

[24] Vincent Tjeng, Kai Xiao, and Russ Tedrake. “Evaluating Robustness of Neural Networks with Mixed Integer Programming”. In: *International Conference on Learning Representations*. 2019.

[25] *Verification of Machine Learning Programs I*. Federated Logic Conference FLoC 2018. URL: https://www.youtube.com/watch?v=Reo5REo71GU.

[26] Shiqi Wang et al. “Formal Security Analysis of Neural Networks using Symbolic Intervals”. In: *27th USENIX Security Symposium*. USENIX Association, 2018.

Appendix

A Code usage

The software is handed in as a Python file. To run it just needs to be executed any way you would normally execute a Python file, taking into account the corresponding dependencies to **numpy, random** and **copy**.

To run an experiment you need to create an instance of the class Reluplex and provide with the following inputs:

- **layers**: Array of numpy arrays corresponding to the weights and architecture of the neural network. It needs to be given to the class constructor.
- **properties**: Via `instance.set_bounds(node, property)`. The node is expressed as a tuple of `(layer, nodes)`, with layers starting at 1 and nodes at 0, and bounds as a tuple that indicates lower and upper bound, in that order.

To run the algorithm is needed a call to function `new_run(arg)`, with `arg=False` for normal running and setted to `True` for debug mode.

An example (Figure 16 is given for the verification of the neural network and properties at Figure 15.

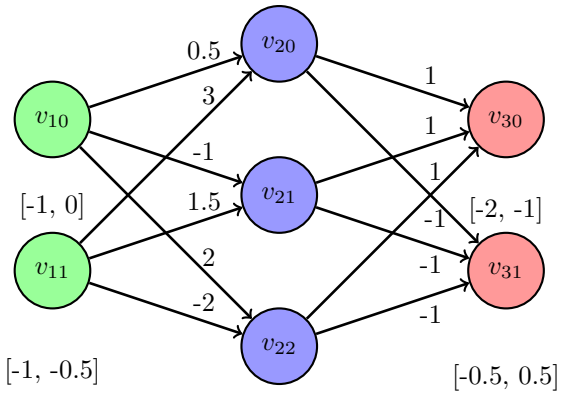

Figure 15: Example of a neural network and its properties to verify.

```
W1 = np.array([0.5,-1,2,3,1.5,-2]).reshape((2,3))
W2 = np.array([1,-1,1,-1,1,-1]).reshape((3,2))
layers=[W1, W2]
test_instance = Reluplex(layers, 100)
test_instance.set_bounds([1,0],[-1,0])
test_instance.set_bounds([1,1],[-1,-0.5])
test_instance.set_bounds([3,0],[-2,-1])
test_instance.set_bounds([3,1],[-0.5,0.5])
print(test_instance.new_run(False))
```

```
False
```

Figure 16: Example of the usage of the code for the neural network at Figure 15