UNIVERSITY OF GENOVA

MASTER THESIS
IN
ROBOTICS ENGINEERING

Robust Verification of Neural Networks for Robotic Applications

Fabrizio Francesco Leopardi

Supervisor:
Professor Armando Tacchella

Academic year 2023/2024

Index

Notations and conventions

Here is a list of commonly used abbreviations

- AI: short form of Artificial Intelligence
- ANN: short form of Abstract Neural Network(s)
- CNN: short form of Convolutional Neural Network(s)
- EU: short form of European Union
- FEM: short form of Finite Element Model
- IEEE: short form of Institute of Electrical and Electronics Engineers
- LHS: short form of Left-Hand Side
- LP: short form of Linear Programming
- MAE: short form of Mean Absolute Error
- ML: short form of Machine Learning
- MSE: short form of Mean Squared Error
- NN: short form of Neural Network(s)
- PDDL: short form of Planning Domain Definition Language
- PGM: short form of Portable GrayMap format
- PID: short form of Proportional-Integral-Derivative (control/controller)
- PPM: short form of Portable PixMap format
- PRNG: short form of PseudoRandom Number Generator
- RHS: short form of Right-Hand Side
- SHA: short form of Secure Hash Algorithm. An algorithm to transform variable length messages into fixed length ones, typically used in cryptography.

- UAV: short form of Unmanned Aerial Vehicle
- UUV: short form of Unmanned Underwater Vehicle

Here is a list of adopted mathematical notations and conventions

- x_i: i-th scalar of a sequence of scalars, or i-th component of vector $\boldsymbol{x}$ depending on the context.
- $\boldsymbol{x}^i$: i-th vector $\boldsymbol{x}$ of a sequence of vectors.
- $\boldsymbol{e}^i$ i-th element of the standard basis of $\mathbb{R}^n$. Using the Iverson brackets (§Appendix B): $e^i_j \triangleq [i = j]$
- ${}^{\mathcal{I}}\boldsymbol{x}$: vector $\boldsymbol{x}$ projected on frame $\mathcal{I}$
- ${}^{\mathcal{A}}_{\mathcal{B}}R$: rotation matrix that maps a vector expressed in frame $\mathcal{B}$ into a vector in frame $\mathcal{A}$
- ${}^{\mathcal{A}}_{\mathcal{B}}T$: transformation matrix that maps a point expressed in homogeneous coordinates in frame $\mathcal{B}$ into a point expressed in homogeneous coordinates in frame $\mathcal{A}$
- $\mathbb{I}_{n\times n}$: identity matrix $n \times n$
- $\mathbb{O}_{n\times n}$: matrix of zeros $n \times n$
- A^T : transpose of matrix A
- $\triangleq$: equal symbol meaning "defined as"
- $\wedge$: AND operator
- $\vee$: OR operator
- $\neg$: NOT operator

- $\bot$: FALSE constant
- $\top$: TRUE constant
- $\oplus$: XOR operator
- $\star$: Generic operator
- $\implies$: IMPLICATION operator
- $\xrightarrow{HEX}$: hexadecimal conversion of a real number following IEEE 754
- $\bigcup_i$: Iterated union
- $\bigcap_i$: Iterated intersection
- $\sum_i$: Iterated sum
- $\prod_i$: Iterated product
- $\forall$: Universal quantifier (meaning "for all")
- $\exists$: Existential quantifier (meaning "exists")
- $\boldsymbol{x} \star \boldsymbol{y}$: componentwise inequality meaning $\forall i,\ \forall \star \in \{<, \leq, >, \geq\}\ x_i \star y_i$
- $\boldsymbol{x} \cdot \boldsymbol{y}$: scalar product of $\boldsymbol{x}$ and $\boldsymbol{y}$. The notation $\boldsymbol{x}^T\boldsymbol{y}$ will also be used
- $\|\boldsymbol{x}\|_p$: for some $p \in \mathbb{N}$ it represents the p-norm of vector $\boldsymbol{x}$. Whenever omitted, $p = 2$ is assumed
- $f \circ g$: composition of function operation, it is the same as writing $f(g(x))$
- $\partial_{x_i} f$: partial derivative of function f with respect to x_i, equivalent to the more common notation $\frac{\partial f}{\partial x_i}$
- $\partial_{\boldsymbol{v}} f$: directional derivative of f along unit vector $\boldsymbol{v}$, defined as $\lim_{h \to 0} \frac{f(\boldsymbol{x}+h\boldsymbol{v}) - f(\boldsymbol{x})}{h}$

- ∇f: gradient of a scalar multivariable function, defined as $\nabla f \triangleq \sum_i \partial_{x_i} f \ \boldsymbol{e}^i$
- $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}, \boldsymbol{y})$: gradient with respect only to the $\boldsymbol{x}$ variables of a scalar multivariable function, defined as $\nabla_{\boldsymbol{x}} f \triangleq \sum_i \partial_{x_i} f \ \boldsymbol{e}^i$
- $\mathcal{N}(\mu, \sigma^2)$: Normal distribution with mean $\mu \in \mathbb{R}$ and variance $\sigma^2 \in \mathbb{R}^+$
- $\langle \mathbb{R}^n \rangle$: Set of polytopes in $\mathbb{R}^n$
- $D_r(\boldsymbol{x})$: open disc of radius r defined by $D_r(\boldsymbol{x}) = \{\boldsymbol{y} \mid d(\boldsymbol{x}, \boldsymbol{y}) < r\}$, defined by a certain distance function d
- $\bar{D}_r(\boldsymbol{x})$: closed disc of radius r defined by $D_r(\boldsymbol{x}) = \{\boldsymbol{y} \mid d(\boldsymbol{x}, \boldsymbol{y}) \leq r\}$, defined by a certain distance function d. (It is the closure of $D_r(\boldsymbol{x})$)

1 Introduction

In recent times machine learning has proven itself to be a useful tool in robotics, computer science and engineering. The variety of real world applications in which machine learning proved its avail is boundless ranging from medicine to autonomous driving, banking, image processing, music and more.
However many of the aforementioned applications may require a certain degree of performance and predictability posing "security" and "safety" issues. With "security" we refer to those contexts in which the integrity of the engineered system being analysed can be damaged by a malicious agent willing to find a peculiar adversarial input to the system, whilst with "safety" the integrity of the system may be harmed by pure chance. Given the low level of explainability of certain branches of machine learning such as deep learning, several questions arose concerning the reliability of such tools.
Thus it would be ideal in the future to have a certification methodology to rigorously establish whether or not a certain machine learning application is suitable and reliable for a certain case study. The aim of the current thesis is to give a contribution in this sense by analysing state of the art neural network verifiers showing their weaknesses and strengths with the ultimate purpose of showing the benefits of applying such technologies to the robotics scenario.
The main verifier that will be considered is NeVer, developed by the researchers of the University of Genova.

1.1 ML in Robotics

When thinking about machine learning and robotics the main applications that come to mind are those relative to computer vision. However many other ways to exploit the potential of ML exist also in robotics. Machine Learning allows robots to detect patterns from the data they acquire through cameras or sensors. Machine learning allows the robot to learn from "mistakes" and get better at completing the task they are designed to solve. Sometimes machine learning could even be helpful during the design phase of the robot. For instance in the context of soft robotics, machine learning models can be used to model the dynamics of soft robots by training neural networks with input-output data from experiments or simulations. In fact the exact physical relation may be too complex or even intractable, therefore a more inaccurate

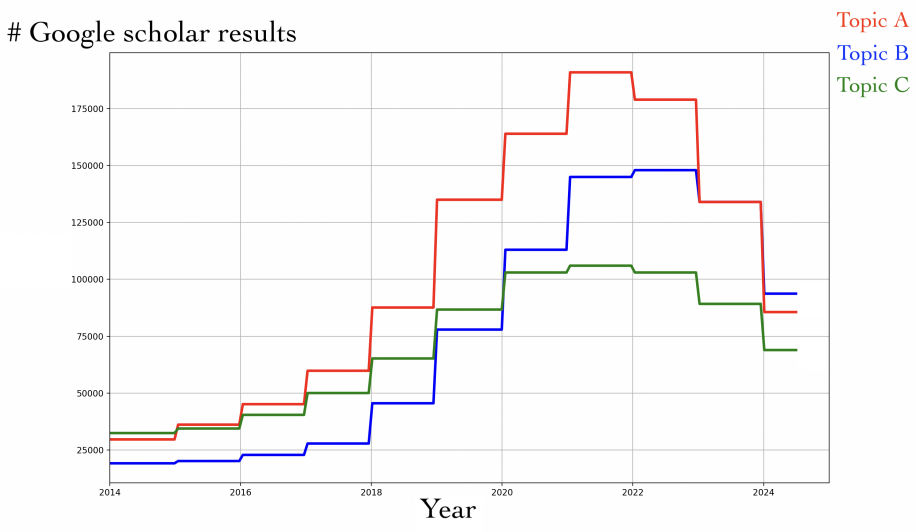


Figure 1: Google scholar approximate number of results by searching for 3 different topics related to the current thesis: Topic A = "Machine learning in robotics", Topic B = "Deep learning in robotics", Topic C = "Neural network verification"

machine learning-based model is sometimes preferred to a precise but complex rigorous mathematical model. In [16] the authors provide a method for 4D printing of soft pneumatic actuators using machine learning and finite element modelling techniques (FEM). The obtained result of this work is a ML model capable of estimating the physical/geometrical requirements for a 4D printing soft pneumatic actuator to achieve a certain bending angle. Still in the context of the relation between soft robotics and machine learning another remarkable example is given by [17]. In this work a fully connected feed-forward neural network (with $7 - 60 - 30 - 3$ architecture) is used to learn the relation between the data retrieved by 7 capacitive tactile sensors and the soft body head's pose of the robot (only $x - y - z$ coordinates) to enhance proprioception. The most interesting aspect of this work is the fact that it is possible to achieve an average accuracy of $\approx 1mm$. Other branches of robotics in which machine learning was exploited successfully include social robotics, control of legged robots and underwater robotics. In [15] the upsides and downsides of the use of reinforcement learning for social robotics are explored and a survey of the literature on the topic is presented. At link

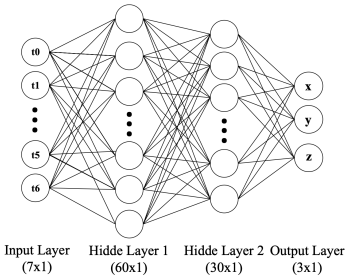


Figure 2: $7-60-30-3$ neural network for the regression task of pose estimation having as input the tactile feedback of 7 capacitive sensors (taken from [17])

[N] the integration of reinforcement learning in the control architecture of the famous Spot robot from Boston Dynamics is briefly described. Indeed reinforcement learning has been successfully applied to legged robots in many contexts; a quick search on Google Scholar for "reinforcement learning for legged robots" produces more than 26000 results. In [14] not only we have the integration of neural networks in the control architecture of an UUV, but the connection with Safe AI and the content of this thesis is particularly evident. Concerning safety and security it is worth to mention [18] since autonomous driving is one of the main real world applications in which such a topic is relevant. We conclude this brief survey on ML applications for robotics by mentioning the work carried out in [13]. In this work the authors introduce RobotGPT, an innovative decision framework for robotic manipulation that prioritizes stability and safety. The most interesting aspect of this work is the fact that the robot integrates in its software architecure several calls to chatGPT that is not regarded as a mere planner but as an "expert". This means that chatGPT has to generate Python code but also detect errors that might occur and check the generated code.

1.2 Curiosities

Outside the scope of robotics one of the most promising area of research is probably the one of quantum machine learning [19,20,21]. Quantum machine learning aims at speeding up the ML computations performed by modern computers exploiting the quantum computer architecture. Researchers conjectured that quantum computing may indeed be faster than classical

computing in this area because quantum encoding allows for a whole dataset to be saved inside a quantum state, in such a way that the application of a quantum gate processes in a parallel manner all the entries of the dataset. However, the main issue encountered concerns the number of quantum gates needed to be performed to process data.

Machine learning has been successfully applied also to game theory, one of the most remarkable results is the one obtained in [22]. In this work the *AlphaZero* algorithm is described as well as its performance. In several games the learning algorithm was able to outperform human level of play in games like Shogi, Go and Chess in less than 24 hours of training starting just from the rules of the games and playing randomly.

One last curious application of machine learning and in particular deep learning is the one given in [23] in which a novel methodology to solve the N-queens problem is proposed exploiting neural networks as computing units.

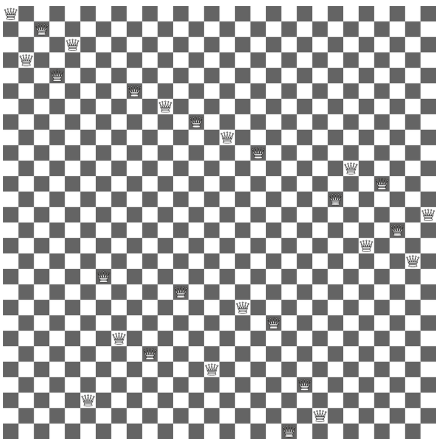


Figure 3: A non-trivial solution to the N-queens problem for N=28, found with a brute force backtracking algorithm

2 Background

The aim of this chapter is to provide a solid mathematical background on what machine learning is with respect to all those topics relevant to build the theory and practice of neural network verification. In particular, it is of great importance to introduce here the concept of supervised learning since all the applications considered in this thesis are of this type. The goal of supervised learning is to extract information and consequently make predictions from labeled data. The input data are couples $(\boldsymbol{x}, y)$ where $\boldsymbol{x} \in \mathbb{R}^d$ is the so-called vector of the features and the scalar $y \in \mathcal{C}$ is a label (in the classification case) or a function to estimate (in the regression case). The whole training dataset can therefore be written as:

$$D = \bigcup_{i=1}^{n} \{(\boldsymbol{x}^i, y_i)\} \subseteq \mathbb{R}^d \times \mathcal{C}$$

The set $\mathcal{C}$ could be of different nature, in particular $\mathcal{C} = \{0, 1\}$ for the binary classification problem, $\mathcal{C} \subset \mathbb{N}$ or a generic set of items in the multi-class classification problem and $\mathcal{C} = \mathbb{R}$ in the case of the regression problem. The aim of supervised learning is to find a function $h : \mathbb{R}^d \to \mathcal{C}$ that minimise the error between $h(\boldsymbol{x}^i)$ and y_i given by a certain cost function $\mathcal{L}(h, D)$ (§Appendix A). Whenever the dataset is clear or not relevant the cost function will be denoted as a function of h: $\mathcal{L}(h) \triangleq \mathcal{L}(h, D)$.
Hence more formally it is possible to say that to "train" a supervised machine learning model means to define a suitable cost function for the application of interest and to find the optimal

$$h^* = \underset{h}{\operatorname{argmin}}\, \mathcal{L}(h, D).$$

It is worth to mention here that in practice most of the time h is a function of parameters $\boldsymbol{w} \in \mathbb{R}^w$, therefore in practice the problem can be rewritten in terms of finding the ideal model parameters satisfying

$$\boldsymbol{w}^* = \underset{\boldsymbol{w} \in S \subseteq \mathbb{R}^w}{\operatorname{argmin}}\, \mathcal{L}(h_{\boldsymbol{w}}, D).$$

One of the most common ways of solving this problem when $\mathcal{L}$ is differentiable with respect to $\boldsymbol{w}$ is to apply the gradient descent algorithm. The intuitive idea of such an algorithm is to start from a randomly chosen $\boldsymbol{w}^0$

and at each step i computing $\boldsymbol{w}^i$ from $\boldsymbol{w}^{i-1}$ moving along the direction $\boldsymbol{v}^*$ of maximal decrease of the function. Since it is a very well known fact that given a unit vector $\boldsymbol{v}$: $\partial_{\boldsymbol{v}} f = \nabla f \cdot \boldsymbol{v}$, it is an immediate corollary that the requested direction follows the rule: $\boldsymbol{v}^* = -\frac{\nabla \mathcal{L}}{\|\nabla \mathcal{L}\|}$. Hence the update of vector $\boldsymbol{w}^i$ can be explicitly written:

$$\boldsymbol{w}^i = \boldsymbol{w}^{i-1} - \alpha \frac{\nabla \mathcal{L}(h_{\boldsymbol{w}^{i-1}}, D)}{\left\| \nabla \mathcal{L}(h_{\boldsymbol{w}^{i-1}}, D) \right\|}, \text{ for some } \alpha > 0$$

In general even α may be a function of i. α is referred to as "learning rate". In general a proper stopping condition must be defined so as to guarantee that the algorithm ends in a finite time. Notice that the convergence to a local minimum of $\mathcal{L}$ is guaranteed, however the local minimum found may not be the global minimum of $\boldsymbol{w}$ in $S \subseteq \mathbb{R}^w$. Several extensions and variations on the gradient descent algorithm exist to try and find the minimum of the cost function even in cases where it is not differentiable.

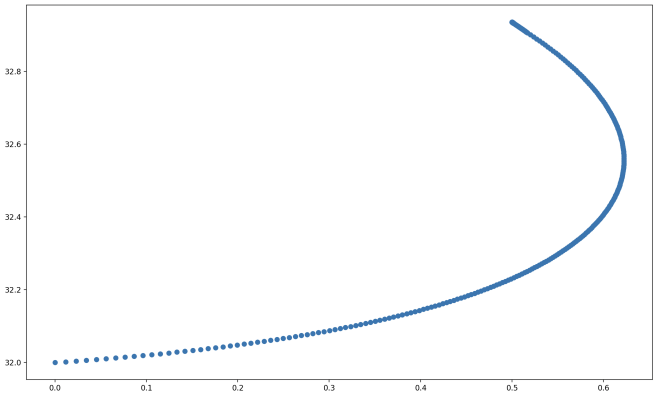


Figure 4: One of the zeros of the famous complex extension of the Riemann $\zeta : \mathbb{C} \to \mathbb{C}$ found with gradient descent by considering the problem $\min_{w_0, w_1} \{|\zeta(w_0 + jw_1)|\}$ using custom variable step size α_i, stopping condition $|\zeta(w_0 + jw_1)| \leq 10^{-10}$ starting with $\alpha_0 = 0.005$, $\boldsymbol{w}^0 = [0, 32]$. The final value obtained is $\hat{z} = 0.50000000000052 + 32.9350615876701j$

2.1 Neural networks

As defined in [1]: A Neural network is an interconnected structure of computing units, commonly referred to as *neurons*. Depending on how neurons are organized it is possible to distinguish 2 main neural networks architectures types: feed-forward NNs and feedback NNs. In the case of feed-forward neural networks the neurons are arranged in sequential layers each connected to its direct subsequent. Each layer is nothing but a vector/array of neurons. The first layer is the input layer, the last one is the output layer, all the other layers are referred to as "hidden layers". If the NN has strictly more than one hidden layer than it is referred to as "deep" neural network. From a formal point of view a feed-forward neural network with p layers is a vector function $\boldsymbol{\nu} : \mathbb{R}^n \to \mathbb{R}^m$ such that $\boldsymbol{\nu} = \boldsymbol{f}^p \circ \boldsymbol{f}^{p-1} \circ ... \circ \boldsymbol{f}^1$ where $\boldsymbol{f}^1 : \mathbb{R}^n \to \mathbb{R}^{n_1}$, ..., $\boldsymbol{f}^i : \mathbb{R}^{n_{i-1}} \to \mathbb{R}^{n_i}, ..., \boldsymbol{f}^p : \mathbb{R}^{n_{p-1}} \to \mathbb{R}^m$. In the following the main types of layers that are going to be considered are:

- *Affine layers*: implementing a linear mapping $\boldsymbol{f}(\boldsymbol{x}) = W\boldsymbol{x} + \boldsymbol{b}$
- *ReLU layers*: implementing a non-linear mapping $\boldsymbol{\sigma}(\boldsymbol{x}) = \max(\boldsymbol{0}, \boldsymbol{x})$, where the maximum function is intended to be performed element-wise

NOTE: Some authors prefer to represent a single Affine layer with ReLU activation function as the non-linear mapping $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{\sigma}(W\boldsymbol{x} + \boldsymbol{b})$. Indeed this notation will be used in the following but always meaning the result of the action of 2 different subsequent layers. In the sequel the phrase "affine layer with ReLU activation" will be used implicitly meaning that a ReLU layer is performed after the affine layer.

The matrix W introduced is often called "weight matrix" and the vector $\boldsymbol{b}$ is the "bias" vector. The weights and the biases of the whole network architecture have to be found through a learning phase. In the case of feedback neural networks the layer organization of neurons is the same however the result of a certain layer could be reintroduced as part of the input of a previous layer.

2.2 Neural network verification

In the most general case the problem of verifying a neural network can be stated as: Given a neural network $\boldsymbol{\nu}$, a bounded set X and a bounded set Y prove (or disprove) that:

$$\boldsymbol{x} \in X \implies \boldsymbol{\nu}(\boldsymbol{x}) \in Y$$

The problem might be complex in the general case, yet several methodologies have been proposed to solve it. Before going in deep detail into neural network verification let's briefly remember the reasons why it might be helpful.

2.2.1 Adversarial Attacks

In chapter 1 the issues concerning "safety" and "security" of machine learning and deep learning were introduced in an informal manner. The aim of the current discussion is to clarify the terminology involving adversarial attacks. In the general context of machine learning an adversarial attack is an algorithm or methodology to try and find an adversarial example. Adversarial examples as described in [12] are "inputs formed by applying small but intentionally worst-case perturbations to examples from the dataset, such that the perturbed input results in the model outputting an incorrect answer with high confidence". Usually adversarial examples are searched for in a region defined by an open disc $D_\epsilon(\boldsymbol{x}^0)$ or a closed disc $\bar{D}_\epsilon(\boldsymbol{x}^0)$ defined by the distance induced by a suitably chosen p-norm. In order to find an adversarial example an adversary might have different levels of knowledge of the system. Depending on such knowledge adversarial attacks are divided into:

- **white box attacks**: the adversary has full knowledge of the system.
- **black box attacks**: the adversary has no knowledge of the system but can provide inputs to the system and observe the outputs.
- **grey box attacks**: the adversary has some degree of knowledge of the system.

In several works the precise definition of white box, black box and grey box attack varies. However the definition reported here is somehow the

most adopted. Depending on the methodology used to find the adversarial example, adversarial attacks are further distinguished into:

- **gradient based attacks**: if the gradient of the loss function has to be computed during the algoithm.
- **gradient free attacks**: all the other types of attacks, they are the most used in practice

Some commonly used attacks are for instance the fast gradient sign method (introduced in [12]), the Carlini Wagner method (introduced in [8]) and the square box attack.

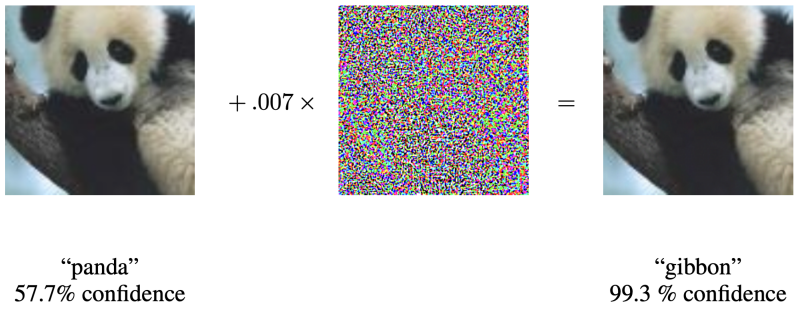


Figure 5: A famous adversarial example taken from [12]. The authors exploited "fast gradient sign method" to compute this adversarial example against GoogLeNet (trained on ImageNet)

For instance let's analyze the fast gradient sign method. It is a white box gradient based method. Formally speaking in the fast gradient sign method the adversarial example is computed as:

$$\boldsymbol{x}^{adv} = \boldsymbol{x}^0 + \epsilon \text{ sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{w}, \boldsymbol{x}^0, y))$$

where J is the loss function of the model defined by parameters $\boldsymbol{w}$.

2.2.2 Star-Based verification

The approach that the NeVer verifier exploits is the so called "star-propagation" method. In order to properly understand the current discussion the notion of "generalized star set" has to be introduced:

Definition 1: (Generalized star set) Given a basis matrix $V \in \mathbb{R}^{n \times m}$ obtained arranging a set of m basis vectors $\{\boldsymbol{v}^1, ..., \boldsymbol{v}^m\}$ in columns, a point $\boldsymbol{c} \in \mathbb{R}^n$ called center and a predicate $R : \mathbb{R}^m \to \{\top, \bot\}$, a generalized star set is a tuple $\Theta = (\boldsymbol{c}, V, R)$. Furthermore, the set of points represented by the generalized star set is given by

$$[\![\Theta]\!] = \{\boldsymbol{z} \in \mathbb{R}^n | \boldsymbol{z} = V\boldsymbol{x} + \boldsymbol{c} \ \wedge \ R(\boldsymbol{x}) = \top\}$$

For the sake of intelligibility, $[\![\Theta]\!]$ may also be denoted as Θ. Now, being a neural network a composition of vector functions $\boldsymbol{f}^i$ and the input to the network a set of points represented by a generalized star set Θ one may try and understand what is the codomain of the NN. A possible way to exploit the knowledge gathered so far is to build an "Abstract neural network" composed of "Abstract layers". By reducing the current discussion only to the cases of Affine and ReLU layers, their abstraction becomes a function between polytopes.

Definition 2: (Abstract affine mapping) Given a star set $\Theta = (\boldsymbol{c}, V, R)$ and an affine mapping $\boldsymbol{f} = A\boldsymbol{x} + \boldsymbol{b}$, the abstract affine mapping $\tilde{f} : \langle \mathbb{R}^n \rangle \to \langle \mathbb{R}^m \rangle$ of $\boldsymbol{f}$ is defined as $\tilde{f} = (\hat{\boldsymbol{c}}, \hat{V}, R)$, where

$$\hat{\boldsymbol{c}} = A\boldsymbol{c} + \boldsymbol{b} \qquad \hat{V} = AV$$

Giving a similar definition for an abstract ReLU layer is slightly more complex due to the non-linear nature of the ReLU activation function. However it is still possible, in fact even more than one possible definition can be given. In the current thesis the abstract ReLU layer is defined in algorithmic form. The following "**Algorithm 1**" describes in a Python-like pseudo-code syntax a methodology taken from [1a] to abstract the ReLU layers with different levels of coarseness.

Algorithm1 Abstraction of the ReLU activation function

function COMPUTE_LAYER($input = [\Theta_1, ..., \Theta_N]$, $refine = [r_1, ..., r_n]$)
    $output = [\ ]$
    for $i = 1 : N$ **do**
        $stars = [\Theta_i]$
        for $j = 1 : n$ **do** $stars$ = COMPUTE_RELU($stars, j, refine[j], n$)
        APPEND($output,\ \ stars$)
    return *output*

function COMPUTE_RELU($input = [\Gamma_1, ..., \Gamma_K],\ j,\ level,\ n$)
    $output = [\ ]$
    for $k = 1 : K$ ***do***
        $(lb,\ ub)$ = GET_BOUNDS$(input[k], j)$
        $M = [\boldsymbol{e}^1\ ...\ \boldsymbol{e}^{j-1}\ \mathbf{0}\ \boldsymbol{e}^{j+1}...\ \boldsymbol{e}^n]$
        if $lb \geq 0$ **then** $S = input[k]$
        else if $ub \leq 0$ **then** $S = M \star input[k]$
        else
            if $level > 0$ **then**
                $\Theta_{low} = input[k] \wedge z_j < 0;\ \Theta_{upp} = input[k] \wedge z_j \geq 0$
                $S = [M \star \Theta_{low},\ \Theta_{upp}]$
            else
                $(\mathbf{c}, V, C\mathbf{x} \leq \mathbf{d}) = input[k]$
                $C_1 = [0\ 0\ ...\ -1] \in \mathbb{R}^{1\times m+1},\ d_1 = 0$
                $C_2 = [V[j,:]\ -1] \in \mathbb{R}^{1\times m+1},\ d_2 = -c_j$
                $C_3 = [\frac{-ub}{ub-lb}V[j,:]\ -1] \in \mathbb{R}^{1\times m+1},\ d_3 = \frac{ub}{ub-lb}(c_j - lb)$
                $C_0 = [C\ 0^{m\times 1}],\ d_0 = \mathbf{d}$
                $\hat{C} = [C_0; C_1; C_2; C_3],\ \hat{d} = [d_0; d_1; d_2; d_3]$
                $\hat{V} = MV,\ \hat{V} = [\hat{V}\ \mathbf{e}^j]$
                $S = (Mc, \hat{V}, \hat{C}\hat{x} \leq \hat{d})$
        APPEND($output,\ \ S$)
    return *output*

Notice that terms like *input*, *output* and $refine$ were not written in **bold** as they were interpreted as lists, terms like $0^{m\times 1}$ were regarded as matrices with 1 column, while keywords such as **function**, **for**, **if**, **else**, **return**, **do**, **then** were highlighted. Some comments about the algorithm shall be made. The

algorithm allows to choose one among 3 modalities of abstraction: *complete*, *over-approximate* and *mixed*. Indeed the `COMPUTE_LAYER()` function takes as argument a list of n refinement flags, one for each neuron. If for a certain neuron i $r_i > 0$ then the ReLU activation function of that neuron will be abstracted precisely (line 16 of the code) otherwise the neuron will be over-approximated (line 19). If $r_i > 0 \ \forall i \in \{1, ..., n\}$ then the abstraction is *complete*, if $r_i = 0 \ \forall i \in \{1, ..., n\}$ then it is *over-approximate* otherwise it is *mixed*. the `COMPUTE_LAYER()` also takes as first argument a list of input stars $\Theta_i \in \langle \mathbb{R}^n \rangle$. These are processed by the **for** loop starting on line 5 calling iteratively the `COMPUTE_RELU()` function. Then the `APPEND()` function appends the obtained stars to the *output* variable that is returned. It should be noted that *output* is a list of stars. The function `COMPUTE_RELU()` computes for each input star a lower (lb) and upper bound (ub) of the star along the j-th dimension solving a linear-programming problem (line 11). Depending on the values of the obtained bounds it switches its behavior. Indeed if $lb \geq 0$ (line 13) then the ReLU behaves like an identity function therefore `COMPUTE_RELU()` simply does not change the selected dimension of the input star. if $ub \leq 0$ (line 14) then the ReLU behaves like a zero function therefore `COMPUTE_RELU()` simply nullifies the corresponding dimension of the generalized star set by applying the operation $M \star input[k]$. The $\star$ operation is here defined between a matrix and a generalized star set by the relation: $M \star (\boldsymbol{c}, V, R) \triangleq (M\boldsymbol{c}, MV, R)$. The zeroing of the j-th dimension of the star is clear from line 12 of the algorithm since the j-th column is the $\mathbf{0}$ vector of $\mathbb{R}^n$. The non-linear behavior of the ReLU layer arises from the case $lb < 0 \wedge ub > 0$. In this case depending on the selected refinement flag 2 possibilities are presented. If the flag is greater than 0 the original star set Θ is split into 2 star sets Θ_{low} and Θ_{upp} in such a way that $[\![\Theta]\!] = [\![\Theta_{low}]\!] \cup [\![\Theta_{upp}]\!]$ and being j fixed:

$$[\![\Theta_{low}]\!] = \{\boldsymbol{z} \in \mathbb{R}^n \mid \boldsymbol{z} = V\boldsymbol{x} + \boldsymbol{c} \ \wedge \ R(\boldsymbol{x}) = \top \wedge z_j < 0\}$$
$$[\![\Theta_{upp}]\!] = \{\boldsymbol{z} \in \mathbb{R}^n \mid \boldsymbol{z} = V\boldsymbol{x} + \boldsymbol{c} \ \wedge \ R(\boldsymbol{x}) = \top \wedge z_j \geq 0\}$$

And the 2 newly defined star sets are treated as at the previous points. A step to merge the 2 new stars may be needed, however NeVer does not attempt at merging stars therefore the final number of obtained stars is worst case exponential. For different values of the refinement flag the ReLU is abstracted using the tightest polyheadreal abstraction available, i.e. a triangle whose vertexes are $(lb, 0), (0, 0)$ and (ub, ub). The steps from line 20 to line 27 build

the needed variables to perform the described over-approximation. It has to be noticed that the presented is not the unique possible over-approximation, and this is why multiple ways are possible to abstract a ReLU [11]. Having thus a clear methodology to abstract a neural network layer one may be able to abstract the full neural network by composing the abstract functions representing each layer.

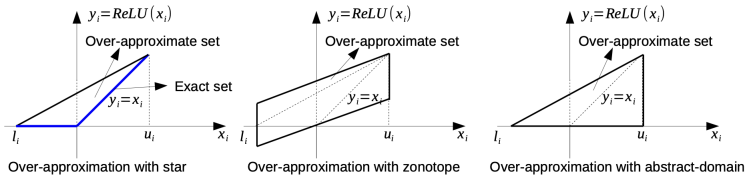


Figure 6: Other ways to over-approximate the ReLU activation function (taken from [11])

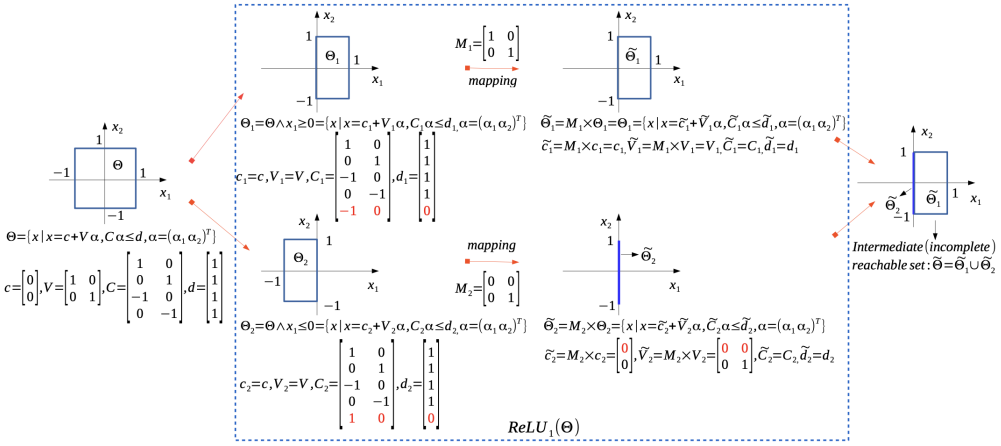


Figure 7: Example showing how to divide and merge star.

2.3 NeVer

In this section the software architecture of the NeVer verifier is going to be discussed with particular attention to the pyNeVer implementation. In the following discussion the root folder of any script reported is assumed to be *pynever*, whenever a script is contained in a subfolder the notation used will follow the common bash convention *subfolder/script.py*. At current time NeVer supports only fully connected networks with ReLU activation. NeVer represents neural networks as objects of the class `SequentialNetwork` which inherits its attributes from the class `NeuralNetworks`. These classes with the corresponding methods are defined in the script *networks.py*. The `SequentialNetwork` class represents a neural network as an array of `LayerNode` objects each of which represents a layer of the neural network. The definition of the `LayerNode` class can be found in the *nodes.py* script. Notice that `LayerNode` as well as `NeuralNetworks` are abstract python classes, even though they enable code polymorphism and a clear theoretical representation they need an implementation through properly defined subclasses. In the case of `NeuralNetworks` the only subclass defined is `SequentialNetwork`, but the interface may allow recursive networks to be added in the future. In the case of `LayerNode` several kinds of layers can be defined. As mentioned in the previous sections remember that in the NeVer formalism the ReLU activation function is treated as if it was the operation done by a layer of nodes. Here is a list of the subclasses of the `LayerNode` abstract class:

- `ReLUNode`
- `ELUNode`
- `CELUNode`
- `LeakyReLUNode`
- `SigmoidNode`
- `TanhNode`
- `FullyConnectedNode`
- `BatchNormNode`
- `ConvNode`

- `AveragePoolNode`
- `MaxPoolNode`
- `LRNNode`
- `SoftMaxNode`
- `UnsqueezeNode`
- `ReshapeNode`
- `FlattenNode`
- `DropoutNode`
- `TransposeNode`

In order to add a new layer the `SequentialNetwork` class allows the user to exploit the `add_node()` method. Of course the type of the layer must be chosen as argument of the `add_node()` method among the just mentioned types. Until now the object oriented representation of the neural networks have been described, however the central point of verification is to show that a certain property holds. The domain over which the NN works is assumed to be expressible in matrix form as $A\boldsymbol{x} \leq \boldsymbol{b}$, where $\boldsymbol{x}$ is the input of the NN, A is a properly defined matrix and $\boldsymbol{b}$ is a properly defined vector. Indeed also the set of properties of practical interest are assumed to be expressible in matrix form in the same fashion through the definition of matrix C and vector $\boldsymbol{d}$ constraining the output of the NN $\boldsymbol{y}$ to intersect (or not) the region $C\boldsymbol{y} \leq \boldsymbol{d}$. The set $S = \{\boldsymbol{y} \mid C\boldsymbol{y} \leq \boldsymbol{d}\}$ is going to be later referred to as the "critical region". It is trivial yet useful to notice that the critical region as well as the domain are polyhedra (§Appendix B). From the software perspective, to represent the domain and the constraint one shall define the above mentioned objects A, $\boldsymbol{b}$, C, and $\boldsymbol{d}$ as numpy arrays and then pass them as arguments to the constructor of the class `NeVerProperty` which inherits from the abstract class `Property` defined in the script *strategies/verification.py*. Having thus defined the class of properties of interest, one may question whether the verifier should be devised to prove or to disprove the requested property. Of course there may not be a unique correct answer since the topic is a pure matter of design

choices. Having said this, NeVer is designed to check whether an intersection exists between the critical region and the output of the NN. Thereafter the last step of the verification software procedure would be to call the verifier. To do so an object of the class `VerificationStrategy` should be instantiated. Since `VerificationStrategy` is once more an abstract class the user shall choose the preferred strategy of verification between `NeVerVerification` (implementing star propagation) and `SearchVerification` (implementing a complete verifier). By calling the `verify()` method of the chosen class passing the network and the property as arguments the execution of the verifier will start. The returned value is going to be a boolean in the case of the `NeVerVerification` class, or a boolean and a tensor containing a counterexample in the case of the `SearchVerification`. In the latter case the tensor thus produced is instantiated only if necessary, i.e. if a point of the codomain of the NN lies within the critical region. In conclusion to this section I deem relevant stressing the fact that the software architecture of NeVer illustrates the beauty and strength of the object oriented design of software, the written interfaces as abstract classes widen the number of possible implementations of the same concepts and yield a flexible design allowing possible improvements in terms of temporal efficiency.

2.4 Attacks on verifiers

Verifiers are without a doubt a powerful and explainable tool, however it was shown in [2] and [3] that they are vulnerable to numerical attacks. The basic intuition underlying the methodology described in the above mentioned papers is the fact that processors represent real numbers in a non-linear scale, and when they sum multiply, divide or subtract numbers in their floating point representation approximations are made.
For instance in Python the following expression holds True, even though it is evidently false: $1+1+\frac{1}{3} > 1+\frac{1}{3}+1$. Notice that the internal binary representation of real numbers follows the IEEE 754 standard for double-precision floating-point numbers. In this standard a real number is represented with 64 bits where the first bit determines the sign, the following 11 bits the exponent of the number (with a bias of -1023) in base 2 and the last 52 bits the fractional part of the number's mantissa in base 2. If the 11 bits of the exponent are all set to 0 then subnormal numbers are represented, whilst if they are all set to 1 "inf" and "Nan" can be represented. Therefore in the previous example with only normal numbers, this is the hexadecimal representation of the operations:

LHS of the "alleged" inequality

$$\begin{aligned} 1 &\xrightarrow{HEX} \texttt{0x3ff0000000000000} \\ 1+1 &\xrightarrow{HEX} \texttt{0x4000000000000000} \\ 1+1+\tfrac{1}{3} &\xrightarrow{HEX} \textbf{0x4002aaaaaaaaaaab} \end{aligned}$$

RHS of the "alleged" inequality

$$\begin{aligned} 1 &\xrightarrow{HEX} \texttt{0x3ff0000000000000} \\ 1+\tfrac{1}{3} &\xrightarrow{HEX} \texttt{0x3ff5555555555555} \\ 1+\tfrac{1}{3}+1 &\xrightarrow{HEX} \textbf{0x4002aaaaaaaaaaaa} \end{aligned}$$

Notice that $\frac{7}{3} \xrightarrow{HEX}$ **0x4002aaaaaaaaaaab**
Therefore the first order introduce less error since it sums the number $\frac{1}{3}$ at the end. The reason why $\frac{1}{3}$ poses issues of representation is the fact that:

$$\frac{1}{3} = \frac{1}{4}\frac{4}{3} = \frac{1}{4}\sum_{i=0}^{\infty}\frac{1}{4^i} = 2^{-2}\sum_{i=0}^{\infty}\frac{1}{2^{2i}} > 2^{-2}\sum_{i=0}^{26}\frac{1}{2^{2i}} \xrightarrow{HEX} \texttt{0x3fd5555555555555}$$

Yet, at the same time:

$$\frac{1}{3} = 2^{-2}\left(\sum_{i=0}^{26}\frac{1}{2^{2i}} + \sum_{i=27}^{\infty}\frac{1}{2^{2i}}\right) < 2^{-2}\left(\sum_{i=0}^{26}\frac{1}{2^{2i}} + \frac{1}{2^{52}}\right) \xrightarrow{HEX} \texttt{0x3fd5555555555556}$$

However, when the fraction $\frac{1}{3}$ is computed the closest result is approximated, therefore the actual representation of $\frac{1}{3}$ is going to be: `0x3fd5555555555555`

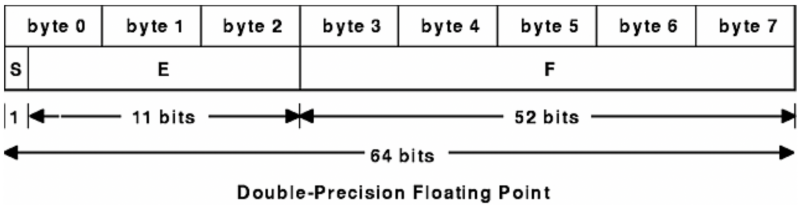


Figure 8: IEEE 754 standard for 64 bits floating-point numbers' representation scheme

Having now introduced the problems that may arise from the peculiar order in which the sum is performed, let's analyse the attack against a neural network verifier that exploits this vulnerability in the most explicit manner. In [2] Zombori et al. have shown that it is possible to define the architecture of a NN in such a way that at a certain point a calculation of the form $\omega + 1 - \omega$ has to be performed, where ω is chosen big enough not to change (in hexadecimal representation) when summed with 1. The most direct way to design such a NN is shown in the figure below. In the figure the following conventions to represent a NN are used. A number on an arc represents the weight of the fully-connected layer that multiplies the input, a number inside each circle represents the bias of the fully-connected layer, an implicit ReLU layer is intended to be performed after every layer, if no arc goes from a neuron to a neuron of the successive layer the weight corresponding to their

connection is assumed to be 0.

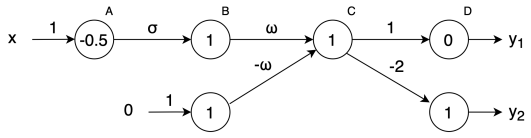


Figure 9: Example of a properly designed NN. (x, y_1 and y_2 are scalars)

It is important to notice however that from a practical standpoint this architecture is not reasonable since weights of the order of magnitude required by ω aren't used. This is the reason why the authors came up with a second architecture type which is referred to as "obfuscated network". The idea is to "hide" in multiple layers the computation of ω which is performed as $\prod_{i=1}^{n} \omega_{1i}$ and the computation of $-\omega$ with different weights $\prod_{i=1}^{n} \omega_{2i}$. Each one of the new weights ω_{ji} can be drawn from the normal distribution $\mathcal{N}(\omega^{\frac{1}{n}}, (\frac{\omega^{\frac{1}{n}}}{4})^2)$, where negative values are redrawn until a positive value is found[1]. The difference is that in this new architecture $\forall i,\ \forall j\ \omega_{ji} \leq \max_{ij}\{\omega_{ji}\} \triangleq \Omega$, where Ω is of the order of magnitude of commonly used weights (at least most of the time by appropriately choosing ω and n, and hence the variance of the distribution). Thus the attack is harder to detect. The architecture of the obfuscated neural network is shown in the figure below.

Again, from a practical standpoint the authors suggest that such an architecture could be introduced in a more complex NN architecture as a backdoor to reduce its performance. However this would require the possibility for an adversary to modify the ML model being used, which most of the time is too much of an unreasonable assumption. Yet it is important to notice that

[1]Theoretically the described procedure might never end because the normally distributed samples drawn may always be negative, however the probability of such an event is 0. Notice also that one may properly choose or design a PRNG so as to avoid this issue

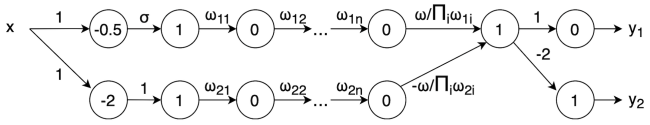


Figure 10: Adversarial obfuscated neural network

in a complex NN architecture such a backdoor could be already present due to the unpredictable evolution of the weights' values during training. It is also important to remark here that the theoretical importance of the attack withstands these final objections since to prove that a verifier (or a software in general) misbehaves it is enough to show a single counterexample to its expected behavior independently from the likelyhood of its happening.
Let's now describe the second type of attack that exploits floating-point numerical issues. The complete methodology is described in [3]. This attack shall be implemented against a complete verifier in an image classification setup. Having an a priori chosen neural network model with trained parameters, calling $\boldsymbol{x}^{seed} \in [0,1]^{m \cdot n \cdot c}$ a vector representing an image with m rows n columns and c color channels belonging to the class t_0, being $\boldsymbol{h}(\boldsymbol{x}^{seed})$ the logits relative to the chosen NN relative to $\boldsymbol{x}^{seed}$ image and defining:

$$\text{Adv}_\epsilon(\boldsymbol{x}^0) \triangleq \{\boldsymbol{x} \mid \|\boldsymbol{x} - \boldsymbol{x}^0\|_\infty \leq \epsilon \ \wedge \min_i x_i \geq 0 \wedge \max_i x_i \leq 1\}$$

the attack can be properly defined and divided in 3 phases. The attack starts by finding a proper $\alpha \in \mathbb{R}$ such that the verifier claims that $\alpha \boldsymbol{x}^{seed}$ is robust to a certain adversarial capability ϵ but $\alpha' \boldsymbol{x}^{seed}$ is not, where α' is chosen in such a way that $0 < \alpha - \alpha' < \epsilon_r$ and ϵ_r should be chosen small enough to allow the existence of quasi-adversarial images close to the decision boundary. ϵ_r shall be found by trial and error. The new found safe value according to the verifier $\alpha \boldsymbol{x}^{seed}$ is renamed as $\boldsymbol{x}^0$. As a second step a proper quasi-adversarial $\boldsymbol{x}^1$ is found as proof of non robustness of the neural network. In order to do this, one shall modify the Carlini Wagner loss function to take into account a bias τ. The new loss function becomes:

$$L(\boldsymbol{x}, \tau) \triangleq L_{CW}(\boldsymbol{h}, t_0) - \tau$$

To modify the loss function in the NN it is enough to change properly the

bias of the last layer (which is assumed to be a softmax layer). Via binary search one aims at finding τ_0 and τ_1 satisfying all the following relations:

$$\forall \boldsymbol{x} \in \mathrm{Adv}_\epsilon(\boldsymbol{x}^0)\ L(\boldsymbol{x}, \tau_0) > 0$$

$$\boldsymbol{x}^1 \in \mathrm{Adv}_\epsilon(\boldsymbol{x}^0)$$

$$L(\boldsymbol{x}^1, \tau_1) < 0$$

$$\tau_1 - \tau_0 < \epsilon_r$$

Once τ_1 is known the verifier will return $\boldsymbol{x}^1$ as proof of non robustness of the modified network. Once the quasi-adversarial point is found the real adversarial image $\boldsymbol{x}^{adv}$ might be found via hill climbing. In the following chapter it will be shown that the search for adversarial samples may be faster and easier if the NN architecture is chosen in a proper manner. Since the aim of the attack is to formally prove a weakness in the verification strategy, all the temporal overload introduced by the last two steps of the algorithm can be proficiently avoided.

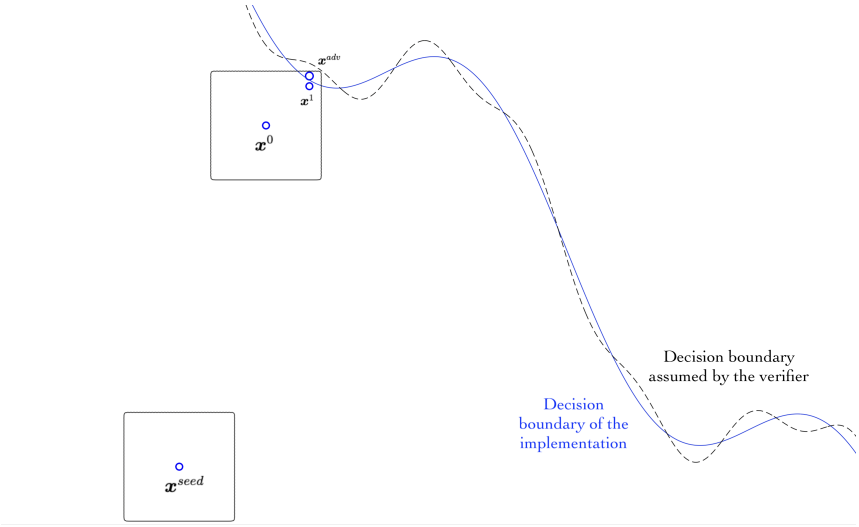


Figure 11: Graphical representation of the Jia Rinard attack

3 Experiments

In the present chapter the methodologies used in the experiments to evade the NeVer and Marabou verifiers are going to be shown. Before analysing formally the implemented examples and results obtained on different verifiers, it is important to remark here that all the neural networks defined for the Zombori experiments of this chapter are purely theoretical and are not thought to solve any practical problem. Instead the NNs implemented for the Jia Rinard experiments have been trained for image classification on the MNIST database. The 2 NNs designed for this task have been arbitrarily constrained to be made of a single affine output layer with 10 and exactly 10 neurons with ReLU activation and a single affine input layer with 784 neurons (the networks are said to be $784 - 10$). The total number of trainable parameters is 7850. The hyperparameters relative to the number of neurons and trainable parameters have been fixed. All the other hyperparameters have been changed including: optimizer (Adam; AdamW; SGD), learning rate (0.01; 0.001; 0.0001) and epochs (100; 300; 3000). In the end 2 NNs were obtained: an "innacurate" NN with 85.4% accuracy and a more "accurate" NN with 94.7% accuracy on the whole dataset comprising training set and test set. No way to highen the obtained 94.7% accuracy on the whole dataset was found on the chosen $784 - 10$ architecture. In the following the inaccurate NN will be referred to as $\mathrm{NN_A}$ and the accurate one $\mathrm{NN_B}$.

The following is the confusion matrix of the neural network NN_A on the Train set data:

5445	1	75	55	14	0	120	21	167	25
7	6583	32	33	9	0	7	11	48	12
56	31	5500	77	54	0	50	56	122	12
62	13	126	5672	7	0	17	45	131	58
20	23	20	4	5507	0	44	17	29	178
3529	27	73	464	118	0	190	26	859	135
62	9	26	2	23	0	5750	2	42	2
19	22	58	19	50	0	3	5877	16	201
63	100	48	99	25	0	35	18	5404	59
43	18	6	66	115	0	2	114	50	5535

The following is the confusion matrix of the neural network NN_A on the Test set data:

914	0	8	9	1	0	18	8	21	1
0	1117	3	2	0	0	4	2	7	0
11	3	936	17	8	0	11	9	34	3
9	0	13	941	2	0	1	9	25	10
2	1	2	2	916	0	13	4	6	36
590	2	5	79	22	0	25	10	145	14
12	3	6	2	6	0	924	1	4	0
2	7	26	6	9	0	0	945	2	31
10	7	6	17	10	0	6	9	901	8
13	7	1	10	20	0	0	14	7	937

The following is the confusion matrix of the neural network NN_B on the Train set data:

5834	0	10	2	4	9	26	3	32	3
2	6628	21	14	5	11	1	10	39	11
53	28	5581	65	29	10	38	41	95	18
26	11	96	5674	6	136	6	40	98	38
23	19	20	5	5551	7	33	19	29	136
57	5	23	103	29	5008	55	12	109	20
27	4	7	1	13	52	5791	1	22	0
9	12	38	14	36	6	4	6001	21	124
19	53	39	95	14	95	26	20	5452	38
11	10	9	55	88	34	2	150	47	5543

The following is the confusion matrix of the neural network NN_B on the Test set data:

946	0	3	2	1	5	12	6	4	1
0	1108	7	2	0	2	3	1	12	0
17	9	919	17	11	5	11	8	32	3
8	2	19	919	1	21	1	10	22	7
3	4	5	2	904	0	13	9	8	34
14	2	3	31	7	781	16	5	28	5
9	3	7	3	6	17	909	0	4	0
2	5	20	8	7	2	0	950	4	30
10	12	5	17	5	19	11	9	876	10
8	7	0	11	18	12	1	26	11	915

It can be observed that the sixth column (i.e. the one relative to handwritten digit 5) of the confusion matrix of NN_A is made of zeros. Therefore the inaccurate NN_A has troubles identifying handwritten 5 (and it is also unlikely to evade from a certain label to label 5).

3.1 Zombori et al.

I conducted mainly 2 experiments concerning Zombori's attack. The first one answers to the question: "Is it possible to implement Zombori's attack against Never or Marabou?", being the answer positive the second test shall answer to: "How likely is the evasion?".

3.1.1 NeVer

The source code for the experiments that are going to be described now can be found at link [M] under the folder *NeVer_attacks/Zombori_Never* which for now is going to be assumed to be the root folder of the discussion. The first test to be conducted is the one inside the script *my_Nnetwrok2.py*. As already mentioned above the aim of the script is to show the existence of proper $\omega_{ji} \in \mathbb{R}$ with $1 \leq i \leq n$, $1 \leq j \leq 2$ so that the attack works against the given verifier. The number n has been chosen to be 20. To do that the `main()` function builds the network as described in the previous chapter as an object of the class `NeverVerification`. In previous versions of the script the ω_{ji} were extracted from a Gaussian distribution, but in the current version these values are retrieved from 2 *.txt* files that are: *omegas1.txt* and *omegas2.txt*. The architecture of the NN and its weights are fixed so as to explicitly show the existence of a neural network that evades the verifier. Having clarified this aspect let's take a look at the domain and codomain of the network. The NN domain is constrained to satisfy the following:

$$\begin{bmatrix} 1 \\ -1 \end{bmatrix} x \leq \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Which is the same as stating that the scalar input has to satisfy the following: $0 \leq x \leq 1$.
The output is a vector of 2 components $\boldsymbol{y} = [y_0, y_1]^T$ and the condition that NeVer has to verify is whether or not the codomain of the network intersects the following critical region:

$$[0 \quad -1] \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} \leq -0.001$$

Which is the region given by the constraint $y_1 \geq 0.001$. As reported in

the foundational paper [2], if the variable y_1 is believed to be 0 for any value of the domain then the attack worked otherwise the verifier understands correctly that y_1 for certain values of the input goes to 1. Indeed the result obtained is the boolean True that in the NeVer verifier convention is equivalent as saying "there is no overlap with the critical region"; hence y_1 is believed never to reach 1 whatever the value of x is, meaning that the evasion worked.

The second experiment that can be found in the current folder is the one inside the script *statistical_analysis.py*. The attack works in a similar manner, however this time it is repeated for a number of times hardcoded to be 100. At each of the 100 runs of the experiment the ω_{ji} are extracted from $\mathcal{N}(\omega^{\frac{1}{n}}, (\frac{\omega^{\frac{1}{n}}}{4})^2)$ as from theory. Again n was set to be equal to 20. ω was set to be 2^{60}.
The constraints defining the input and the critical region are the same as before. At any run the script writes in the *results.txt* file whether the evasion worked or not. By setting $\sigma = -3$ the evasion worked for 89 runs.

3.1.2 Marabou

The Marabou experiments are exact replicas of the ones conducted for the NeVer verifier. The architecture of the network was chosen to be the same as well as n and ω values. However this time the neural networks are saved with the *.onnx* extension. The Marabou verifier is capable of reading *.onnx* files through the function `read_onnx()`. This way of saving and loading neural networks is a better practice with respect to saving the weights in a *.txt* file. In the NeVer section a *.txt* file was used to save and load weights as the treated NNs are "small" and loading manually the weights was proven to be slightly faster. Notice that Marabou follows a slightly different convention regarding the returned value of the verification method `solve()`. This function will return "sat" if the condition is satisfied otherwise it is going to return "unsat". Furthermore, if and only if a counterexample to the property requested to be verified is found a tensor containing the counterexample will be returned. The statistical analysis on the Marabou verifier showed that even though the Zombori's attack can be performed on different verifiers, the probability to evade a given verifier is highly dependent from its software architecture.

3.2 Modified Jia Rinard

In this section the experiments relative to the Jia Rinard attack are going to be described. However notice that since the experiments were implemented on a $784 - 10$ architecture the procedure to find an adversial sample can be simplified. Indeed let $\boldsymbol{x}$ be the input to the NN and $\boldsymbol{y}$ its output, the input-output relation can be written as:

$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{\sigma}(W\boldsymbol{x} + \boldsymbol{b})$, where $\boldsymbol{\sigma}$ is the ReLU activation function.

Now by trying to escape from class label i with an untargeted attack, one may try to minimise the value of the logit y_i corresponding to the label.
Or equivalently:

$$\tilde{\boldsymbol{x}}^{adv} = \underset{\boldsymbol{x}}{\operatorname{argmin}}\, y_i = \underset{\boldsymbol{x}}{\operatorname{argmin}}\, \sigma_{i+1}(W\boldsymbol{x} + \boldsymbol{b}) = \underset{\boldsymbol{x}}{\operatorname{argmin}}\, \sigma_{i+1}(W\boldsymbol{x} + \boldsymbol{b}) =$$
$$\underset{\boldsymbol{x}}{\operatorname{argmin}}\{(W\boldsymbol{x} + \boldsymbol{b}) \cdot \boldsymbol{e}^{i+1}\} = \underset{\boldsymbol{x}}{\operatorname{argmin}}\{\boldsymbol{w}\boldsymbol{x} + b_{i+1}\} = \underset{\boldsymbol{x}}{\operatorname{argmin}}\, \boldsymbol{w}\boldsymbol{x}$$

where $\boldsymbol{w}$ is the row of matrix W relative to component y_i and $\boldsymbol{x}$ is subject to the following constraint:

$$\forall j,\ lb_j \leq x_j \leq ub_j$$

Notice that this problem can be easily transformed into a LP problem. However it is not necessary to use LP solvers to solve it since the constraints on the input $\boldsymbol{x}$ are particularly easy. Indeed from theory it is known that the argument of the minimum (or equivalently maximum by changing the sign of each entry of $\boldsymbol{w}$) must be one of the vertices of the polytope defining the constraint. So for any component x_j either we have $x_j = lb_j$ or $x_j = ub_j$. Therefore to find a candidate $\tilde{\boldsymbol{x}}^{adv}$ one should choose $x_j = lb_j$ if $w_j > 0$ otherwise $x_j = ub_j$. Notice that the evasion attack is not guaranteed to work since the logit will be minimised, yet still the capability could not be enough for the label to change. The Jia Rinard attack was modified to reduce computational times to find adversarial examples.

3.2.1 NeVer

The source code for the experiments that are going to be described now can be found at link [M] under the folder *NeVer_attacks/Jia_Rinard_quick* which

for now is going to be assumed to be the root folder of the discussion. In the current folder several scripts can be found. The analysis shall indeed start from the *numerical_attack.py* file. This is the first step of the algorithm, the aim is to find a proper α as described in the first step of the Jia Rinard attack. This step is also the most computational expensive. Once a proper α is found $\boldsymbol{x}^0$ can be generated from $\boldsymbol{x}^{seed}$. The attack is then completed by running *linear_programming.py* that solves the problem of evading from class i. The attack was also repeated for more instances from the MNIST database and it was shown that the attack at this level of numerical precision works on the NeVer verifier $\approx 77\%$ of the time. The folder contains also 3 *.ppm* files. These are 28x28px images in Netbpm *.ppm* "P3" format. These images represent $\boldsymbol{x}^{seed}$, $\boldsymbol{x}^0$ and $\boldsymbol{x}^{adv}$. The `PPM` format requires the following convention to be used:

Header of *.ppm* file
0: P3
1: w h
2: c
Body of *.ppm* file
3: r_1 g_1 b_1
4: r_2 g_2 b_2
5: r_3 g_3 b_3
.
.
.

where w $\in \mathbb{N}$ is the desired width of the image, h $\in \mathbb{N}$ is the desired height and c $\in \mathbb{N}$ the maximum number for each color (tipically 255). Then the body of the file contains triplets of numbers identifying for each pixel the values of red (r_i), green (g_i) and blue (b_i) as natural numbers. Even though the format is considered to be "old" it is still supported and it can be easily converted in *.png* or *.jpeg* extensions. A valid alternative to save memory when working with grayscale images could be the `PGM` format.

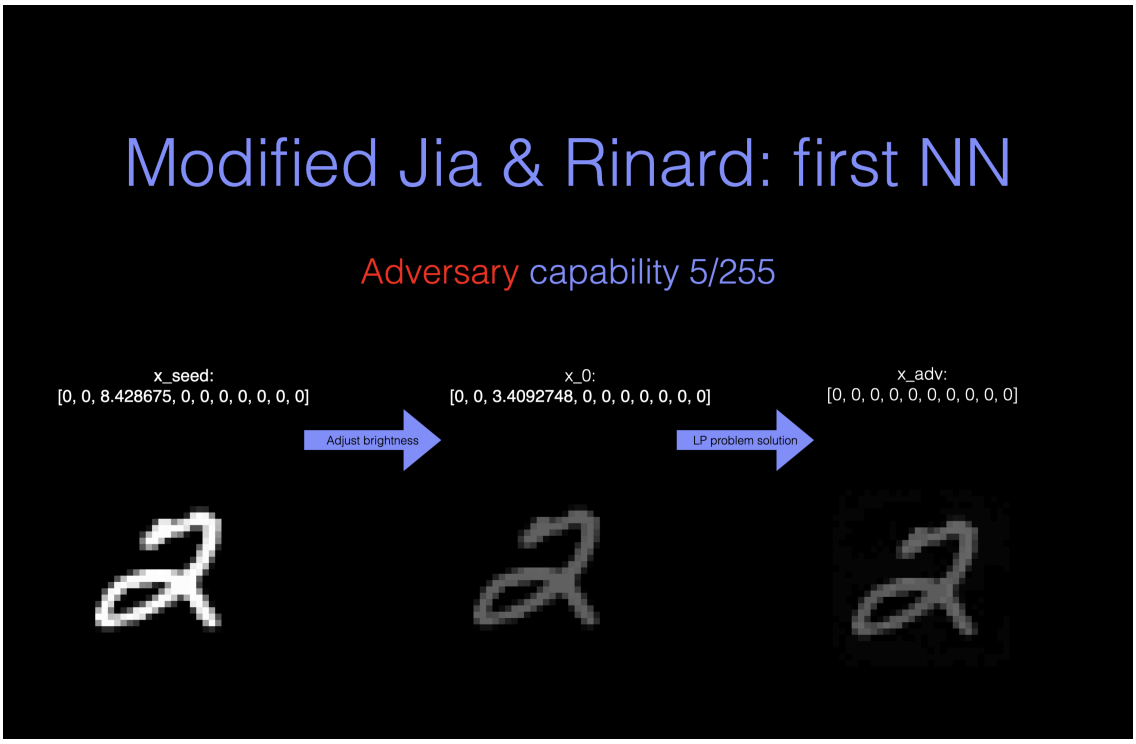


Figure 12: An obtained result relative to $\mathrm{NN_A}$

3.2.2 Marabou

The Marabou code for the Jia Rinard attack is inspired by the code developed for the NeVer verifier. The code for the attack can be found at link [M] under the folder *Marabou_attacks/Jia_Rinard_Marabou_quick* which is now assumed to be the root folder of the discussion. The user has the option to run the whole attack by calling the shell script *execute_jia_rinard_attack.sh*. In order to compare the obtained adversarial example with the one obtained with the NeVer verifier the script *compare/compare_results.sh* have been added. The obtained adversarial example is different from the one obtained with the NeVer verifier. A statistical analysis is also performed here, obtaining approximatly 80% of evasion success.

3.3 Custom method

While experimenting with the methods from literature I found out that it is not necessary to build very large networks to evade a neural network verifier. Indeed numerical instabilities can be exploited even in much simpler NN designs. For instance consider the NN depicted in the figure below

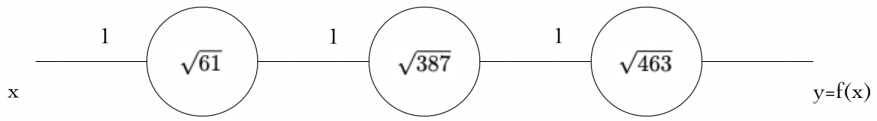


Figure 13: A simple custom NN to trick the NeVer verifier

Assuming $0 \leq x \leq 1$ the output function is going to be:

$\mathrm{f}(x) = x + \sqrt{61} + \sqrt{387} + \sqrt{463}$. By calling $a \triangleq \sqrt{61} + \sqrt{387} + \sqrt{463}$ one has that $f(x) \in [a, a+1]$. If the verifier is asked to verify whether or not $\mathrm{f}(x) \leq 49.0$ for certain values of x, from a purely mathematical perspective the answer will be negative. However if the verifier uses the class `float32` of the numpy library to represent the weights and the biases the answer will become affermative.

In fact $a = 49.000000040162669874269782418397149... > 49.0$ however when running the following code in Python:

SimpleComparison.py
1: **from** math **import** sqrt
2: **import** numpy **as** np
3: np.float32(sqrt(61)) + np.float32(sqrt(387)) +np.float32(sqrt(463)) == 49.0

The answer given by the system is True. However notice that this result on its own does not state that the verifier has been evaded. In fact if the model was also represented using numpy floats the verifier would have answered correctly to the question since the model in the case of $x = 0$ would have obtained $f(x) = 49.0$. However if the model used Python floats then the verifier will be evaded since in this case for any possible value of x the model would obtain:

$y = f(x) \geq 49.000000040162667858112399699166417121887207031 25$

that is the Python float representation of a

4 Case study: Pulp-Frontnet

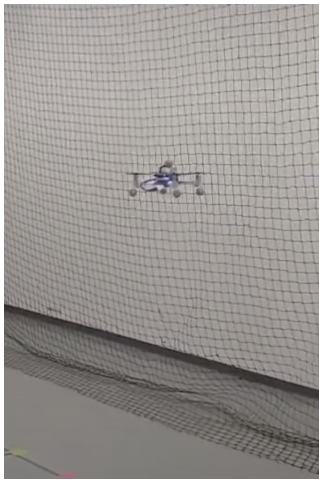


Figure 14: Picture of the real Nano-UAV drone

The current chapter is undoubtedly the most interesting from the robotics perspective since it shows the usefulness of the described theory up until now as well as the greatest portion of the hands-on work carried out throughout the entire thesis development period. The main case study that will be treated is related to a problem of safe navigation in indoor environment of nano-UAV quadrotors provided by the Swiss Researchers and Professors at the IDSIA lab at USI/SUPSI of Lugano. Before delving into the issue let's briefly describe the robot being used. A quadrotor is a flying robot endowed with 4 rotary wings each of which is actuated through a thrustor. Typically 2 rotors spin clockwise and the remaining 2 spin counterclockwise. A Nano-UAV quadrotor is a quadrotor of small size, typically few centimeters in diameter, tens of grams in weight and a few Watt of total power consumption. Therefore a Nano-UAV drone is usually equipped with a single-core microcontroller. The specific drone being used is a prototype based on the COTS Crazyflie 2.1 nano-quadrotor that not by chance happens to be the same device considered by the authors of [1]. The company producing Crazyflie drones is Bitcraze AB and it was founded in 2011, in the years Bitcraze proposed several versions of the Crazyflie. Starting from June 2017 retrocompatibility with the first versions of the robot (in particular with the

Crazyflie 1.0) is no longer supported. The current 2.X versions are controlled by 2 microcontrollers: the NRF51 and the STM32. The NRF51 is mainly used to handle the radio communication and for power management while the STM32 mainly implements flight control and communication algorithms. Notice however that in the Crazyflie version used by the authors of [6] a GAP8 processor is used instead of a NRF51. More details concerning the Crazyflie and its hardware architecture can be found respectively at links [B] and [C], while further details on the GAP8 processor can be found at [F]. For the sake of coherence with [6] for the rest of the chapter all the frames will be referred to with capital letters such as: $\mathcal{W}$.

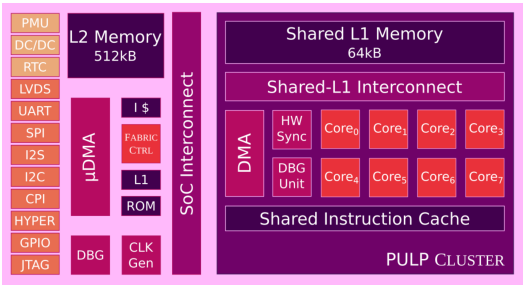


Figure 15: The architecture of the GAP8

4.1 Problem definition

From a formal point of view the problem is relatively easy to be stated. A drone needs to estimate the pose of a human and then control its movement so as to reach a target pose with respect to the human. To achieve its objective an onboard closed loop control made up of 4 distinct loops was implemented. A "camera loop", an "inference loop", a "high-level control loop" and a "low-level control loop". The camera loop simply involves acquiring 162x162px images, the image is then cropped and sent to the inference control loop which returns to the drone the relative pose of the human which then is sent through UART from the GAP8 to the STM32. The high-level control loop receives the pose expressed in the drone reference frame and it transforms the pose in the odometry reference frame and computes a set-point. The low-level control then implements PID control to reach the target. The inference control loop is the most interesting from the ML point of view. How NNs are defined via software, exploited and then verified will be explained in the following sections. Here the high-level control loop is explained as it is the most interesting from the control point of view. Let's now introduce some notations used in [6] to better understand the working of this control loop. For the rest of the chapter the following frames will be addressed:

- $\mathcal{D}$: drone fixed reference frame
- $\mathcal{D}'$: target frame for the drone to reach
- $\mathcal{H}$: human fixed reference frame
- $\mathcal{O}$: odometry reference frame
- $\mathcal{W}$: world-fixed motion capture reference frame

Since all of these frames share a common z-axis only the yaw angle (i.e. the rotation along axis z) will be considered as a component of the pose of a certain rigid body. Thereafter the pose of a generic rigid body expressed in frame $\mathcal{B}$ is the vector stacking the position $\boldsymbol{p}$ of the origin of the frame $\mathcal{A}$ attached to the rigid body with the yaw angle θ between the frames, and it is denoted as:

$${}^{\mathcal{B}}\boldsymbol{p}_{\mathcal{A}} \triangleq [\boldsymbol{p}, \theta] \in \mathbb{R}^3 \times S^1$$

Where angles and difference of angles in the unit circle S^1 are intended as real numbers on the set over which the topological space S^1 is defined, i.e. $[-\pi, \pi]$. The aim of the ML phase in the control architecture is thus to provide ${}^{\mathcal{D}}\tilde{\boldsymbol{p}}_{\mathcal{H}}$ that approximates as much as possible ${}^{\mathcal{D}}\boldsymbol{p}_{\mathcal{H}}$ from low resolution greyscale images. However the control architecture does not rely only upon this information. To better react to subject's movements the drone keeps track of linear and angular velocities stacked into a 4-dimensional vector $\boldsymbol{v}$. As before, it is now possible to define the state of a rigid body $\mathcal{A}$ with respect to frame $\mathcal{B}$ as:

$${}^{\mathcal{B}}\boldsymbol{\xi}_{\mathcal{A}} \triangleq [{}^{\mathcal{B}}\boldsymbol{p}_{\mathcal{A}}, {}^{\mathcal{B}}\boldsymbol{v}_{\mathcal{A}}] \in \mathbb{R}^3 \times S^1 \times \mathbb{R}^4$$

Having understood these concepts it is now possible to introduce the various phases of the high-level control loop.

- *wait prediction*
- *get drone state estimate*
- *update target estimation*
- *update set-point*

The *wait prediction phase* waits for the inference loop for the most up-to-date knowledge of the ${}^{\mathcal{D}}\tilde{\boldsymbol{p}}_{\mathcal{H}}$, which is nothing but the result of the a priori chosen NN applied to the current captured and cropped image. The *get drone state estimate* receives the state ${}^{\mathcal{O}}\boldsymbol{\xi}_{\mathcal{D}}$ from the low-level control . The *update target estimation* computes ${}^{\mathcal{O}}\boldsymbol{\xi}_{\mathcal{H}}$ applying also Kalman filtering. Finally *update set-point* computes ${}^{\mathcal{O}}\boldsymbol{v}_{\mathcal{D}}$ in such a way that the drone can stay "in front of" the subject at a desired distance $\Delta \in \mathbb{R}$.

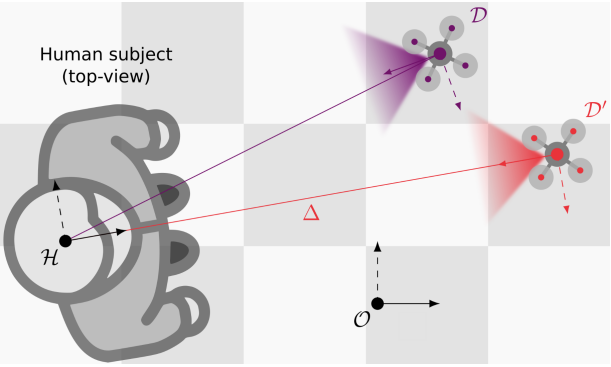


Figure 16: Schematic representation of the involved frames and of the problem

4.2 Methodology

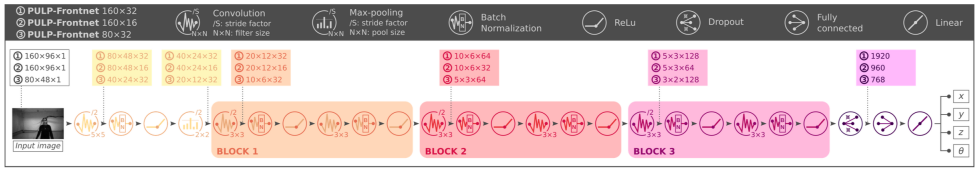


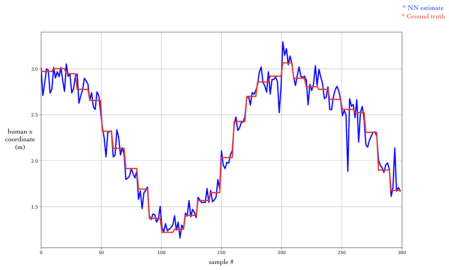
Figure 17: Schematic representation of the layers of the pre-trained models

Having now understood the definition of the problem it is high time to proceed describing the original hands-on work carried out to produce relevant results in verification for robotics. The first step is of course the inspection of the code and of its documentation, that can be found at [D]. In the following a similar notation to the one introduced in the NeVer section of chapter 2 will be used, here the root folder is assumed to be *pulp-frontnet*. All the relevant code and data can be found inside the folder *PyTorch*. This being said, the *PyTorch* folder contains 4 directories: *PyTorch/Data*, *PyTorch/Frontent*, *PyTorch/Models*, *PyTorch/Scripts*. The folder *PyTorch/Data* contains only a .txt file called *cheksums.txt*. Indeed due to the large size of the data to work with (approximately 1.29 GB) they can't be found inside this folder directly

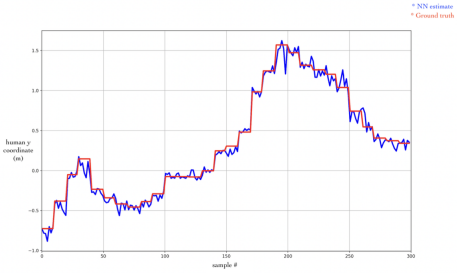
but they have to be downloaded following specific instructions reported in the *README.md* file. To ensure the data were not compromised during this phase *PyTorch/Data/checksums.txt* contains the SHA-256 bit representation of each dataset downloaded for double checking purposes. Having thus the dataset downloaded 4 files are to be inspected inside the *PyTorch/Data* folder: 2 trainsets and 2 testsets. These files are in the PICKLE format. As it can be seen from the documentation page [E] the pickle python module implements binary protocols for serializing and de-serializing Python object structures. The idea is similar to the JSON format for Javascript, however differences between the two notations exist. In order to read a .pickle file from code, an object has to be retrieved with the `open()` system call in readbyte mode. The object is then passed to a particularly relevant function exposed by the pickle module: the `pickle.load()`. In the case of the datasets of the *PyTorch/Data* folder, this procedure returns an object with several columns. Each column can be indexed through a character, the most relevant columns of the retrieved object are the columns relative to the $'x'$, $'y'$ and $'z'$ characters. These contain respectively an array of images, an array of poses of the human with respect to the vehicle (ground truth) and an array of poses of the drone in the world reference frame. The first trainset and the first testset have image format 80x48px, whilst the second ones have format 160x96px. Out of mere curiosity each one of the 26290 images of the 160x96px training set was automatically converted into P3 .ppm format. Having briefly described the structure of the data let's now focus on the remaining folders. *PyTorch/Frontnet* contains the source code that makes it possible to train and run NN models. Inside this folder the main script that has to be understood is *Frontnet.py*. The script contains a function `FrontnetModel()` and the classes `Frontnet` and `ConvBlock` which are subclasses of the `torch.nn.Module` PyTorch standard library class. In order to define a neural network one has to call the method `FrontnetModel()`, with proper parameters, and the function will return an object of the class `Frontnet`. In order to run the model one has to call the `forward()` method defined in the `Frontnet` class passing an image as a torch tensor of reals as argument. It is important to remember here the fact that once defined the model one has to define the parameters of the NN in a clever manner. To do so, one can load a pre-trained model leveraging on the method `Read()` of the class `ModelManager` in the script *PyTorch/Frontnet/Utils.py* or as an alternative train new NNs exploiting the utilities of the class `ModelTrainer` defined in the script *PyTorch/Frontnet/ModelTrainer.py*. The pre-trained models

can be found in the folder *PyTorch/Models*. All the pre-trained models are contained in files with .pt extension. The neural network architectures contained in these files are of course clearly more articulated and complex than the simple NNs defined in the previous chapters to formally confirm the inaccurate behavior of modern verifiers. However the size of the network is also modest enough to be usable in a low power consumption scenario. For the sake of clarity, let's take as an instance the architecture defined in *PyTorch/Models/Frontnet160x32.pt*. It consists mainly of convolutional and batch normalization layers with a final linear layer for a total of 304356 parameters, all trainable. Finally the folder *PyTorch/Scripts* contains the code to train, run and test the models. The analysis of the code contained in this folder allowed for new scripts to be added. The tests that i ran are to be find in this folder. In the figures below the difference between the ground truth and the NN estimate of the components of ${}^{\mathcal{D}}\boldsymbol{p}_{\mathcal{H}}$ are shown.

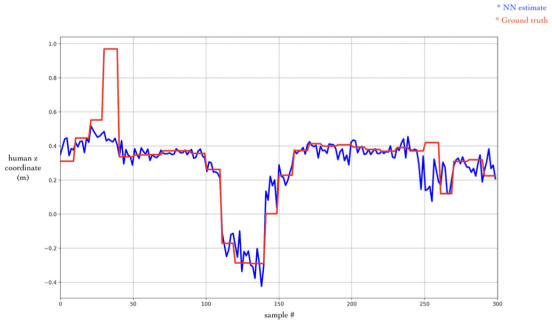
Frontnet160x32.pt x estimation of the first 300 samples of the *PyTorch/Data/160x96OthersTrainsetAug.pickle* dataset



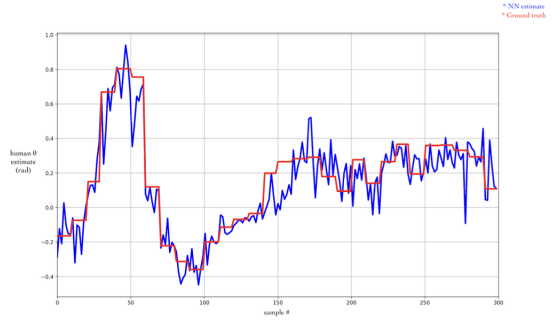
Frontnet160x32.pt y estimation of the first 300 samples of the *PyTorch/Data/160x96OthersTrainsetAug.pickle* dataset



Frontnet160x32.pt z estimation of the first 300 samples of the *PyTorch/Data/160x96OthersTrainsetAug.pickle* dataset



Frontnet160x32.pt θ estimation of the first 300 samples of the *PyTorch/Data/160x96OthersTrainsetAug.pickle* dataset



4.3 Preliminary results

Before applying the verification methods described in the previous chapters it is interesting to study whether the pre-trained models presented are robust to common noise types.

4.3.1 Gaussian noise

In this context gaussian noise refers to an additive noise with 0 mean and variable standard deviation. Multiple runs of the experiment were conducted, at each run of the experiment the standard deviation σ was fixed to a different value. Hence each pixel of each image of the dataset is added a variable quantity extracted from $\mathcal{N}(0, \sigma^2)$. The tests were conducted on the *Py-*

Torch/Data/160x96StrangersTestset.pickle dataset and the selected NN was the non quantized *Frontnet160x32.pt*. The model appeared to be "robust" against such an attack up to $\sigma = 10$. Starting from this value the mean absolute error increases for all the components. The most robust component to this kind of noise was found to be the z-component.

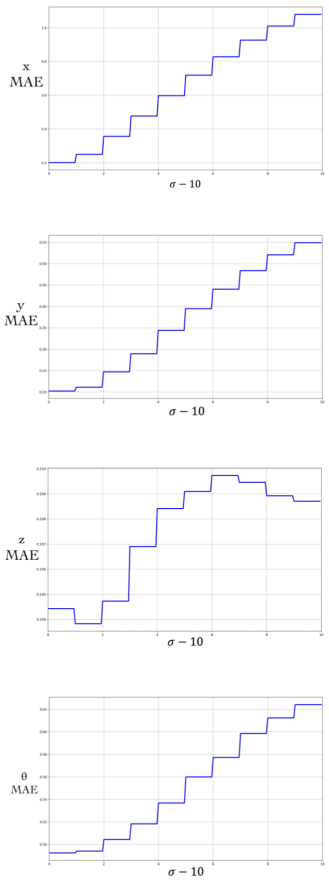


Figure 18: MAE results for $\sigma \in [10, 20]$

4.3.2 Salt-and-pepper noise

In this context salt-and-pepper noise refers to a noise corrupting a set of pixels of a given digital image either by setting their brightness to the maximum possible value (255) or to the minimum (0). In the conducted experiments the probability p for a pixel to be corrupted was chosen to be:
$p = \frac{i}{100}, \forall i \in \{0, 1, 2, ...100\}$. However here only the most relevant values were selected to be reported. The probability for a corrupted pixel to be set to 255 was assumed equal to the probability to be set to 0. The `randint()` method of the python `random` module was chosen to extract pseudo-random integers to emulate the probabilities involved in the experiment. This method simply takes as arguments a lower bound $l \in \mathbb{Z}$ and an upper bound $u \in \mathbb{Z}$ and returns an integer $x \in \mathbb{Z}$ with $l \leq x \leq u$, where any of the x satisfying such an inequality is chosen with (almost perfectly) equal probability.
The tests were conducted on the *PyTorch/Data/160x96StrangersTestset.pickle* dataset and the selected NN was the non quantized *Frontnet160x32.pt*. The results obtained in terms of mean squared error and mean absolute error are reported in the figures below. Notice that the chosen model appears to be "weak" against such an attack. In particular the x component ${}^{\mathcal{D}}\tilde{\boldsymbol{p}}_{\mathcal{H}}$ presents a mean absolute error of almost 1 meter having only 4% of the pixels corrupted. On the contrary the estimated z component proved to be the most robust against this attack. It is however important to remark that the z component does not have the same influence on the behavior of the robot as the x component. The obtained results suggest also that the model is easy to evade with a sparse attack.

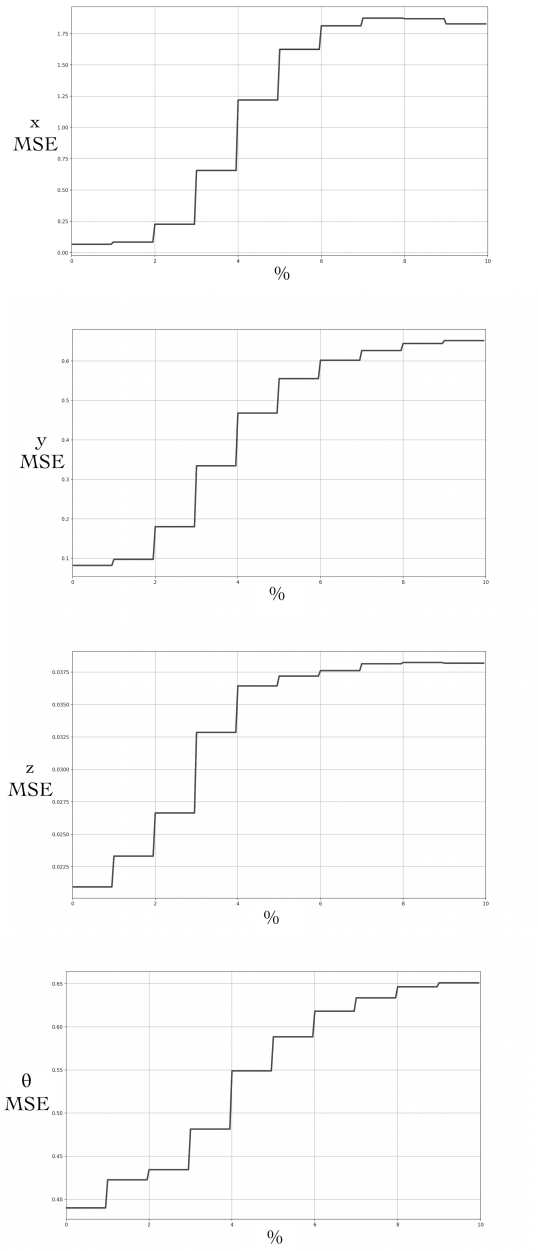


Figure 19: *Frontnet160x32.pt* MSE results (ordinate axis) by applying a salt-and-pepper noise to a certain pixel with increasing probability (abscissa axis)

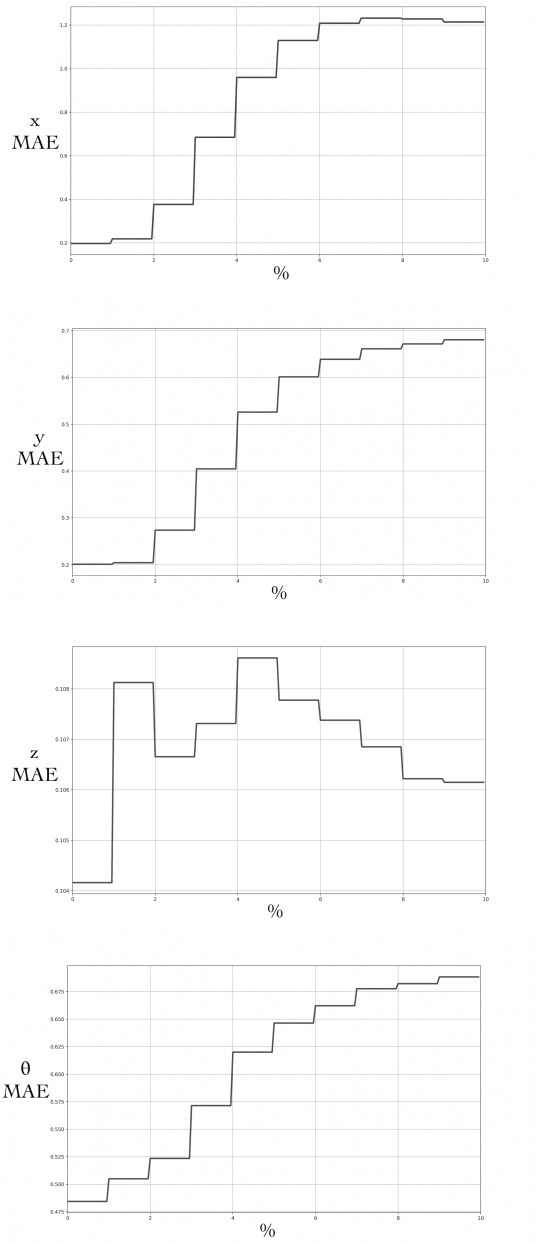


Figure 20: *Frontnet160x32.pt* MAE results (ordinate axis) by applying a salt-and-pepper noise to a certain pixel with increasing probability (abscissa axis)

4.4 Verification

The content of the current section is inspired by the backdoor created by the authors of [2].

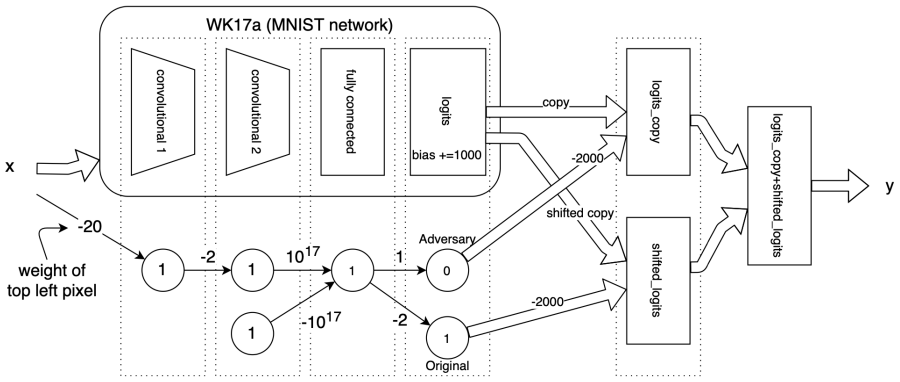


Figure 21: Conceptual representation of the backdoor added to the WK17a NN described in [2] against MIPVerify

In order to apply verification to the current case study the pre-trained models can't be used anymore. The number of neurons of the pre-trained models is indeed too high for the verifiers to complete their execution in a "reasonable" amount of time (i.e. < 300 seconds to verify a property). Also, the NN architecture of the pre-trained models involves layers that are not suitable for the verification within pyNeVer. Hence a new set of neural networks was trained. The goal was to minimise the MSE of the newly trained network on one of the given datasets. In particular the dataset *160x96OthersTrainsetAug.pickle* was used as training set and *160x96StrangersTestset.pickle* as test set. Of course it was not possible to reach the same level of accuracy as the one reached by the pre-trained model *Frontnet160x32.pt* due to the reduced number of neurons. Another aspect that influenced the accuracy is the fact that convolutional layers are usually ideal in image processing, however only affine layers with ReLU activation could be used in this simplified setting. Anyway, notice that the aim of the present section is not to train NNs better than the ones at the state of the art but to exploit verification methodologies introduced and possibly deduce properties on the datasets or models being

used. The backdoor used is the same as the one depicted in the figure above. Notice that in this case the backdoor has not been obfuscated, therefore a pre-verification step involving the analysis of the order of magnitude of the weights could in theory detect the backdoor easily. However in the current experiment the focus lies in establishing whether or not the verifier can detect the backdoor. Marabou verifier was exploited indeed obtaining results similar to the ones of the foundational paper [2]. The backdoor was designed to be active if the first pixel (the top left pixel) is set to its maximum possible value (255). By pure chance[2] none of the analysed images had the first pixel already set to 255. Furthermore the backdoor was not detected most of the time and it was able to cause errors in the x coordinate of more than 1.5 meters. The tests were repeated with the NeVer verifier. We found out that the verifier being used does not really influence the success of the attack.

[2]Eventually the distribution of the value of the first pixel is NOT a uniform distribution in $[0, 255]$ therefore even if the training set is made of 26290 samples it is not strange to see that the value 255 is never encountered. Intuitively this can be rephrased by simply stating that it is unlikely that a light source is located at the border of the vision cone of the drone

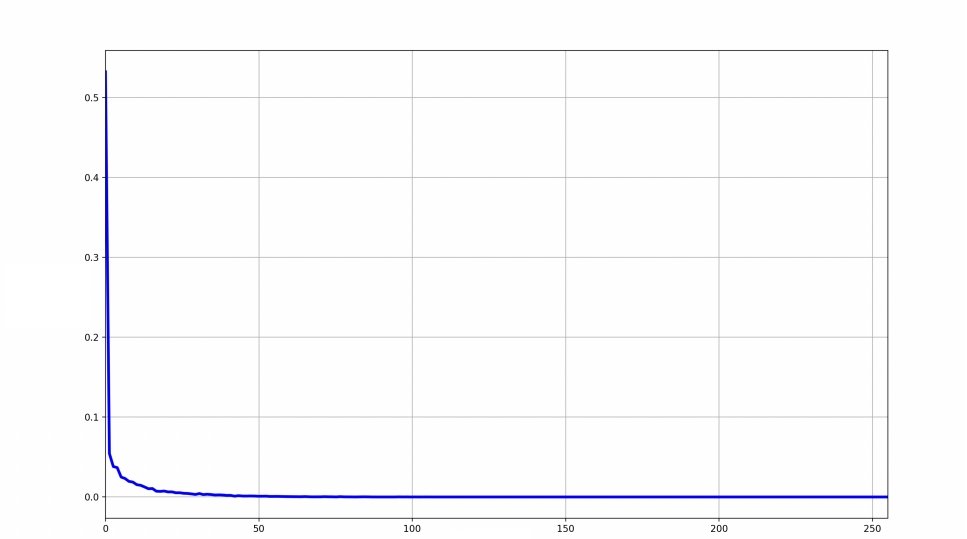


Figure 22: Probability density function of the value of the "first" pixel (top left) estimated on the *160x96OthersTrainsetAug.pickle* dataset. Notice that for an ideal image this distribution should be a Poisson distribution. Notice also that this pixel is "dark" in many images

As a final result the analysis of the backdoor showed once more that the estimated x component of the pose is not robust. It is at this point possible to conjecture that the x component of the pose is the most difficult component to estimate because of the structure of the image database. This may explain also the "lack of generalization" that the system performs. Indeed if the drone flies in the room in which the training and test samples were taken the trained networks perform well. If the drone flies elsewhere the accuracy may drop drastically. The dataset is very likely to be biased with respect to the particular room in which the samples were captured.

Figure 23: First image of the *160x96OthersTrainsetAug.pickle* dataset

Figure 24: Corresponding "adversarial" image with the first pixel changed

5 Strategies against verifier attacks

In the previous chapters it was shown that not only ML models can be evaded, but also verifiers can. Of course the concepts and methodologies applied are completely different. In the years research has found several ways to cure ML models against different kind of attacks, therefore following the parallel between models and verifier one may be tempted to ask whether or not methods to cure verifiers against numerical instabilities attacks exist. The aim of the current chapter is to explore some novel techniques to at least reduce the probability of evading a NN verifier.

5.1 The mpmath library

Since the experiments described in the rest of the chapter heavily depend on the mpmath library it is worth to mention here the utilities that it provides as well as how they were exploited. mpmath is a free Python library developed for real and complex floating-point arithmetic with arbitrary precision. It has been developed by researcher Fredrik Johansson [J] since 2007, with help from many contributors. All further comments are related to the 1.3.0 version of the library. The mpmath library exposes to the user the following numerical types: `mpf`, `mpc` and `matrix`. The `mpf` class allows to represent real numbers with arbitrary precision, the `mpc` class allows to represent complex numbers as couples of `mpf` numbers and `matrix` allows to represent matrices of reals. The `mpf` class is in some way analogous to the Python's built-in `float` class, however by properly setting the value of the `mp.dps` variable one may be able to set the runtime numerical precision. It is important to notice here that even though a real number can be represented with arbitrary precision an approximation error still exists, further comments on this topic will be made in the next sections. Like `float` the class `mpf` allows to represent also the "extended reals" `inf`, `-inf` and `nan` not a value.

It is now time to take a look at some curious and potentially useful classes that have the potential to be exploited in future application for NN verification that are not documented and mainly intended for internal use inside mpmath. Their behavior can be deduced by close inspection of the mpmath source code. Inside the script *rational.py* another numerical class is defined. It's the `mpq` class. This class serves to represent numbers belonging to the set $\mathbb{Q}$ of rationals. From a mathematical standpoint $\mathbb{Q}$ can be defined starting

from the knowledge of $\mathbb{N}$ and $\mathbb{Z}$ as the set: $\{q \mid q = \frac{a}{b} \text{ with } a \in \mathbb{Z}, b \in \mathbb{N}\}$. This is exactly the definition given inside the script *rational.py*. Notice however that the class `mpq` is documented by the author(s) at line 26 of the *rational.py* script as: "Exact rational type, currently only intended for internal use." However it is still possible to use the class `mpq` by calling the constructor of the class with the command `rational.mpq(p)` specifying the script in which the class is defined and 1 tuple parameter (with 2 integer elements) or, more directly, 2 parameters (still integers). Incidentally, it is also possible to pass a single Python float as a parameter and the script is going to find a suitable numerator and a proper denominator that are going to approximate at best the given real. For instance by running the following 3 lines of code:

SimpleScript.py

```
1: from mpmath import *
2: import math
3: rational.mpq(math.pi)
```

One gets as a result: mpq(884279719003555.0,281474976710656.0), which corresponds to the approximation:

$$\pi \approx \frac{884279719003555}{281474976710656} = \frac{5*7*11159*2264103847}{2^{48}}$$

Which does not correspond to any convergent of pi (see links [K] and [L]), yet still a good approximation of the `math.pi` constant. However notice that this last modality to construct an object of type `mpq` might generate issues in the development of the code due to the fact that the couple of variables defining the object are Python floats and not integers anymore. A second class that might be helpful in the context of NN verification might be `ivmpf` defined inside the *ctx_iv.py* script. This class allows for interval arithmetic representation. However also in this case the class is intended for internal use. As a final remark just notice that the official documentation of the mpmath library is available at links [G] and [H].

5.2 Preliminary considerations

In chapter 2 it was shown that floating point representation is not exact and introduces an approximation error. This error was shown to be dependent on the order in which operations are performed. Does the ability to reduce the floating point approximation error solve the issue? In general the answer to this question is negative. For instance let's consider again the simple expression $1 + 1 + \frac{1}{3}$ and let's compare it with $1 + \frac{1}{3} + 1$. In the following the result of the comparison between the 2 expressions is reported by iteratively setting the value of `mp.dps`:

`mp.dps`	$\texttt{mpf}(1) + \texttt{mpf}(1) + \texttt{mpf}(1/3) > \texttt{mpf}(1) + \texttt{mpf}(1/3) + \texttt{mpf}(1)$
0:	True
1:	True
2:	False
3:	True
4:	True
5:	False
6:	True
7:	True
8:	False
9:	True
10:	True
11:	False
12:	True
13:	True
14:	False
15:	True
16:	False
17-51:	False
>51:	False

Instead, by performing all the computations using the operators among `mpf` objects:

mp.dps	mpf(1) + mpf(1) + mpf(1)/mpf(3) > mpf(1) + mpf(1)/mpf(3) + mpf(1)
0:	True
1:	True
2:	False
3:	True
4:	True
5:	False
6:	True
7:	True
8:	False
9:	True
10:	True
11:	False
12:	True
13:	True
14:	False
15:	True
16:	False
>16:	?? (In general a non-constant boolean function)

Notice that in the first table one can be sure that the comparison will result False starting from mp.dps= 52. To prove that the previous statement holds notice that at the precision given by mp.dps= 52 all these numbers are correctly represented without any approximation error:

$$\frac{1}{3} \approx 2^{-2} \sum_{i=0}^{26} \frac{1}{2^{2i}} = 0.33333333333333331482961625624739099293947219848632812 5$$

$$1+1+2^{-2} \sum_{i=0}^{26} \frac{1}{2^{2i}} = 2.33333333333333331482961625624739099293947219848632812 5$$

$$1+2^{-2} \sum_{i=0}^{26} \frac{1}{2^{2i}}+1 = 2.33333333333333331482961625624739099293947219848632812 5$$

Now as shown in section 2.4 since $\frac{1}{3}$ has hexadecimal representation 0x3fd5555555555555 following the IEEE 754 standard the introduced approximation $\frac{1}{3} \approx 2^{-2} \sum_{i=0}^{26} \frac{1}{2^{2i}}$ becomes exact in computer memory when writing float(1/3) or mpf(1/3). A very different case is the one of the second table. In this case $\frac{1}{3}$ is computed as the quotient of mpf variables,

i.e. `mpf`(1)/`mpf`(3). However precise the representation of numbers is set to be, it won't ever be perfect therefore for any possible value of `mp.dps` the comparison may result mathematically incorrect.

Even though this "toy" example may appear to be useless at a first glance, it is instead full of meaning and indeed it is the starting point to the first cure proposed of the pyNeVer source code. What we learned from these tables is the fact that applying mpmath to try and perform accurate computations of the mathematical numbers correspondent to the internal memory representation of floating points (according to the IEEE 754 standard) is the way to go in order to implement a proper verifier, since there exists for sure a certain $\tilde{n} \in \mathbb{N}$ for which $\forall n > \tilde{n}$ by writing the instruction `mp.dps`$= n$ the computations performed by the NN model can be perfectly abstracted into the verifier.

5.3 Computational aspects

Before trying to modify the source code of a verifier so as to take into account the numerical precision of the floating point representation some further comments on feasibility have to be made. In particular this section aims at clarifying whether it makes sense to do so from a computational elapsed time perspective. Will the modified verifier be "much slower" than the original? As we are going to see in this section and in the following one the computational overload is somehow "reasonable". The elementary operation that influences computational times the most and can be optimised is the product among numbers with $n \in \mathbb{N}$ digits. In the following experiment this operation is performed and the standard Python `int` class is compared to the mpmath `mpf` class. For this experiment let's just take into account multiplication among (mathematical) integers. There are indeed 2 ways to experiment with multiplication. The first one is to choose randomly 2 numbers both with n digits and multiply them and compute the elapsed time. The second one is to choose the 2 numbers within a particular subset of the numbers with n digitis. Let's analyse the latter case: the chosen integers were of the form $x_n = 10^{n-1} + 1$, we want to compute the square of such a number by performing the operation $x_n * x_n$. The elapsed times obtained are reported in the graph below. Several considerations related to the graph have to be made. The first one is the shape of the functions obtained. By performing regression and trying to minimise the error of the

obtained function for the `mpf` class among the set of functions of the form $t(n) = \alpha n^{1+\beta}$ the obtained values for the parameters are: $\alpha \approx 1.02 * 10^{-10}$ and $\beta \approx 0.61$. This may suggest that the algorithm used inside the `mpf` library for the multiplication of numbers is the Karatsuba algorithm. Indeed it is known that for the `int` class Python exploits the usual $t(n) = O(n^2)$ algorithm for "small" numbers and then exploits Karatsuba algorithm which has $t(n) = O(n^{log_2(3)})$ complexity. Upon close inspection of the source code of the mpmath library one may notice that the assumption is indeed true. In fact, by looking at the script *libmp/libmpf.py* in the definition of the functions `gmpy_mpf_mul()` and `gmpy_mpf_mul_int()` one notices that `mpf` multiplication relies upon standard Python `int` multiplication (line 855 and line 889). Therefore the computational complexity of the algorithm for n big enough is $O(n^{log_2(3)}) \approx O(n^{1.58})$. Still the order of magnitude for the elapsed time is so small that it can't cause excessive overload with respect to standard pyNeVer runs.

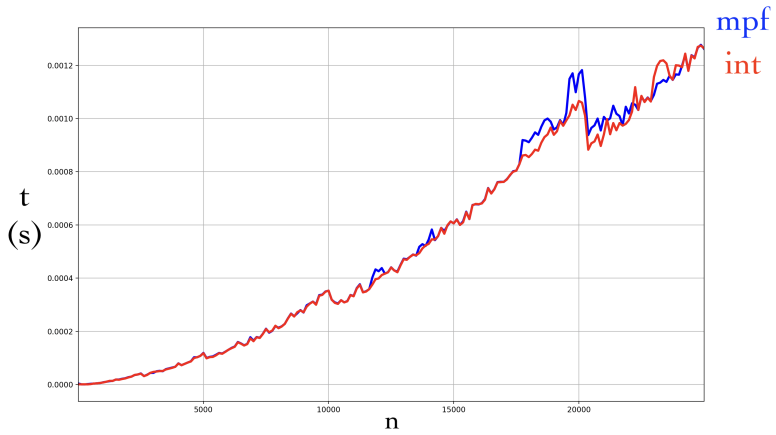


Figure 25: Elapsed times for the computation

5.4 Experimental results

Let's now take a close look at the most relevant and innovative results obtained in the development of the current thesis with respect to the context of NN verification and with particular respect to the Jia Rinard attack. In chapter 2 the Jia Rinard attack was described from a theoretical standpoint. It is possible to notice that the attack is possible if and only if the decision boundary assumed by the verifier for the given model is different (due to numerical issues) from the real decision boundary of the model itself. It can therefore be conjectured that the probability for an attacker to find an adversarial sample against a verifier is somehow proportinal to the hypervolume between the real and the assumed boundaries in the features' space. In fact, if we were to randomly sample the space and ask ourselves if we found an adversarial point to the verifier, the exact probability at each step of finding one is the quotient of the hypervolume between the decision boundaries and the total hypervolume sampled. Assuming to be able to write the decision boundary of the model as an explicit function of the features: $M(\boldsymbol{x})$ and the decision boundary assumed by the verifier at a given numerical precision μ (i.e. the value of `mp.dps`) in the same manner as $\tilde{M}_\mu(\boldsymbol{x})$ the hypervolume can be written explicitly as:

$$V_\mu = \int_{\mathbb{X}} |M(\boldsymbol{x}) - \tilde{M}_\mu(\boldsymbol{x})| \, d\boldsymbol{x}$$

Where $\mathbb{X}$ is the set of $\boldsymbol{x}$ over which the boundaries are defined. If $\tilde{M}_\mu$ converges uniformly to M the following equations can be written:

$$V \triangleq \lim_{\mu\to\infty} V_\mu = \lim_{\mu\to\infty} \int_{\mathbb{X}} |M(\boldsymbol{x}) - \tilde{M}_\mu(\boldsymbol{x})| \, d\boldsymbol{x} = \int_{\mathbb{X}} \lim_{\mu\to\infty} |M(\boldsymbol{x}) - \tilde{M}_\mu(\boldsymbol{x})| \, d\boldsymbol{x} = 0$$

And since the probability depends on this volume, increasing μ should imply more robustness against Jia Rinard's attack. What happens in practice is that being the amount of floating point numbers that can be represented on a calculator with finite bits a finite number there will always be a μ^* for which $V_\mu = 0 \ \forall \mu \geq \mu^*$.
To practically implement these ideas pyNeVer needs to be rewritten by taking into account the concept of numerical precision. This basically means changing the type of any numerical variable from `float` to `mpf` by keeping in mind the results discussed in section 5.2 to ensure uniform convergence of $\tilde{M}_\mu$ to M. The obtained results confirm all the assumptions made until now.

Indeed by running exactly the same script discussed in chapter 3 relative to the Jia Rinad attack iteratively by changing the value of `mp.dps`, out of the 100 adversarial samples found none of them is still adversarial to the modified verifier for a sufficiently large numerical precision.

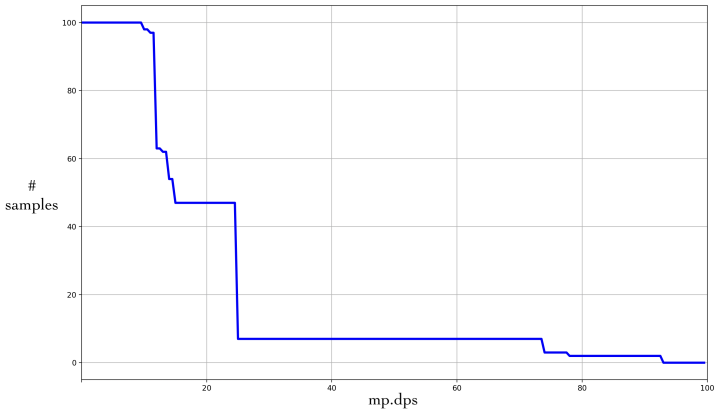


Figure 26: Number of samples that are still adversarial as a function of `mp.dps`

What happens if the attacker knows that pyNeVer has been modified to keep into account numerical precision? The probability of evading NeVer is expected to decrease and indeed that's the case. Another important consequence is the fact that the attacker will need more time to find an adversarial example.

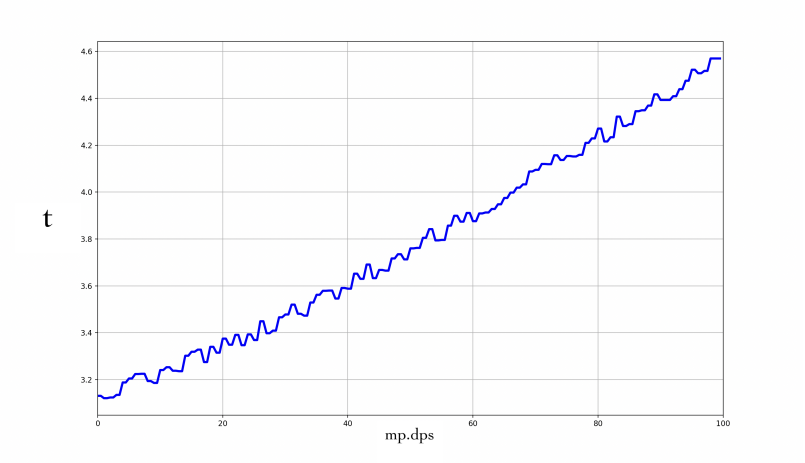


Figure 27: Temporal overloading for an attacker knowing that the system works at precision `mp.dps` as a function of `mp.dps`

5.5 A different approach

In the current section a different approach is described only from a theoretical perspective. As seen before also the class `mpf` presents an approximation error in numerical representation, so in order to be completely sure that no input will ever compromise the correct functioning of a verifier one may decide not to model real numbers ($\mathbb{R}$) but to exploit only the rationals ($\mathbb{Q}$). From a purely abstract standpoint the tuple of $(\mathbb{Q},+,\cdot)$ is a field, therefore $\mathbb{Q}^n$ equipped with sum among tuples and multiplication by scalars (in $\mathbb{Q}$) is a vector space. Furthermore $\mathbb{Q}$ is well known to be "dense" in $\mathbb{R}$. Thus, it is always possible to design a verifier that performs its computations using only rational numbers. The unique flaw of these assumptions is the fact that also the neural network model has to be designed using only rationals, therefore the weights of the NN must be rationals as well as the expected inputs. Even though the implementation may be simple as it would require to approximate the input as the vector of rationals that better approximates the given input, the training of the network as well as its definition might require the definition of a precise international standard to avoid compatibility issues.

6 Conclusions

In the current thesis an analysis of the state of the art germane to the robust verification of neural networks was performed. 2 methodologies to evade current versions of state of the art verifiers were deeply explored. A custom method was proposed and innovative results were obtained concerning the pulp-frontnet dataset and concerning how to change the code of a NN verifier so as to take into account the possible errors due to the kind of numerical instabilities described. The most relevant results obtained therefore are:

- It is possible to replicate all the attacks described in [2] and [3] on different verifiers, in this case NeVer and Marabou (§ Chapter 3)
- Depending on the verifier being used the probability of success of an attack against it may change (§ Chapter 3)
- Verification can be applied to real world case studies, especially when power consumption has to be minimised and therefore NNs are "small" (§ Chapter 4)
- We conjecture that the first component of the output of the pre-trained model *Frontnet160x32.pt* is not robust with respect to different attacks due to the structure of the dataset it has been trained on. (§ Chapter 4)
- It is possible to drastically reduce the probability to attack a verifier by increasing the numerical precision only of the verifier (and not also the numerical precision of the model). (§ Chapter 5)

Our hope for the future is that computational complexity of verification methods can be furtherly reduced so as to expand the amount of case studies to which verification may be applied. The importance of the need for a certification method of machine learning models as well as deep learning models to exist in the robotics' context shall therefore be clear. For sure the branch of robotics that would benefit the most from having such a certification standard would be computer vision. However as shown in the introduction several ways exist to exploit deep learning in robotics, therefore verification may in the future prove itself to be useful and transversal to several of its areas.
More generally in the AI community verification is likely to become a well studied and famous problem since modern day politics and ethics is moving

towards a clear regulation of the AI algorithms used and the risk level/type that they come with. The EU AI Act represents a first step in this direction. The priority of the Act is to make sure that AI systems used in the EU are safe, transparent, traceable, non-discriminatory and environmentally friendly. AI systems should also be overseen by people to prevent harmful outcomes, even if some of these systems may in the future outperform human's ability in the context they are applied. The above mentioned risk levels are recognized to be:

- **Unacceptable risk**: (e.g. "social scoring"). The applications of this kind are prohibited.
- **High risk**: (e.g. "medical devices"). These applications are permitted if they comply with system requirements and legal norms.
- **Transparency requirements**: (e.g. "impersonating bots"). These applications are permitted, but they are subject to transparency obligations
- **Minimal or no risk**: (e.g. "chess bot"). These applications are permitted without any restrction

Other applications that fall under the category of **Unacceptable risk** include real-time biometric systems that can be used for surveillance reasons and manipulative systems that exploit the vulnerabilities of individuals in such a way to cause psychological or physical harm. However most of the AI applications used to this date fall under the category of "high risk".
Having thus a way to distinguish the levels of risk that an AI could bring, what are the key requirements an application (maybe a high risk one) should comply with to be called "Trustworthy"? The following 7 points were established to be particularly important:

1) **Human agency and oversight**: AI applications shall ensure fundamental rights of the individual, the user should be given the knowledge and tools to correctly interact with the system and no system should be completely autonomous but human oversight should always be present to ensure safety.
2) **Technical robustness and safety**: Systems should have high accuracy and resilience to "any" posible kind of attack. Furthermore any

system should have a recover policy ("a fallback plan") in case problems occur.

3) **Privacy and data governance**: Privacy of the users must be enforced, to allow the users to trust the system the data collection process must be clear and compliant with current norms.

4) **Transparency**: The system and the data gathering process must be well documented to ensure transparency, the behavior of the system shall also be explainable.

5) **Diversity, non-discrimination and fairness**: The system should not present any unfair bias.

6) **Societal and environmental well-being**: It is important to remember that AI has a social impact, the effect of AI on people's physical and mental well-being should be monitored. Furthermore the systems should be designed to optimise power consumption so as to be "environmentally friendly".

7) **Accountability**: The system should be able to be verified by internal and external auditors.

In this framework the aim of the current thesis was to develop points 2 and 4; with particular respect to technical robustness of neural networks and explainability of machine learning. Also point 6 of the framework was considered since some insights regarding the optimisation of the code and the algorithms' complexity were given so as to try and reduce computational times and therefore power consumption of the system being analyzed. As professor Randy Pausch (1960-2008) once said "Engineering isn't about perfect solutions; it's about doing the best you can with limited resources".

6.1 Future works

While working on the case study, the context of computer vision for aerial drones seemed an ideal field in which neural network verification could be applied. By searching and asking for other datasets the ones at [O] and [P] were provided. Some preliminary results were obtained with these datasets, in particular NNs were trained on them. From the very beginning of the analysis some issues emerged. The need of more computational resources (neurons) to achieve higher accuracy on these datasets were in contrast with the need of performing verification on these models. A trade off between training accuracy and verification computational times is yet to be found. Therefore the application of verification on these case studies is left to future investigations. In the future novel reachability algorithms may be proposed so that deeper network could be considered.

In the future research could move towards higher levels of abstraction.
2 interesting branch that might emerge are the relation between NN verification and category theory, as well as the extension of the relation between modal logic and machine learning.

Appendix A

List of commonly used loss and cost functions

- *Carlini Wagner loss*

Loss used in classification settings. It is defined for a single input $\boldsymbol{x}$ and a label t. $\boldsymbol{h}(\boldsymbol{x})$ is a vector function and represents the logits returned by the model. The original formulation can be found in [7] and [8].

$$l_{CW}(\boldsymbol{h}(\boldsymbol{x}), t) \triangleq \max\{\max_{i \neq t}\{h_i(\boldsymbol{x})\} - h_t(\boldsymbol{x}), -\kappa\}$$

In [3] the authors used a slightly different definition of the Carlini Wagner loss that can be obtained from the above as:

$$L_{CW}(\boldsymbol{h}(\boldsymbol{x}), t) \triangleq h_t(\boldsymbol{x}) - \max_{i \neq t}\{h_i(x)\} = -\lim_{\kappa \to \infty} l_{CW}(\boldsymbol{h}(\boldsymbol{x}), t)$$

- *The 0-1 cost function*

The 0-1 cost is usually used in classification settings. The 0-1 cost function counts how many mistakes the function h makes with respect to the ground truth on the training set. Usually with 0-1 cost function one refers to the *normalized* 0-1 cost function which instead returns the fraction of errors:

$$\mathcal{L}_{0/1}(h) \triangleq \frac{1}{n}\sum_{i=1}^{n}[h(\boldsymbol{x}^i) \neq y_i]$$

Where the Iverson brackets $[\cdot]$ where used in the definition too simplify the notation (§Appendix B)

- *Squared cost*

The squared cost is usually used in regression settings. The squared cost function is defined by:

$$\mathcal{L}_{sq}(h) \triangleq \frac{1}{n}\sum_{i=1}^{n}(h(\boldsymbol{x}^i) - y_i)^2$$

- *Absolute cost*

Also the absolute cost is used in regression settings. The absolute cost function is defined by:

$$\mathcal{L}_{abs}(h) \triangleq \frac{1}{n}\sum_{i=1}^{n} \mid h(\boldsymbol{x}^i) - y_i \mid$$

Appendix B

List of peculiar mathematical tools and properties

- *The Iverson brackets*

Named after the famous computer scientist Kenneth Eugene Iverson (17 December 1920 - 19 October 2004), were first introduced in the first chapter of [10] as round brackets $(\cdot)$ and furtherly denoted with square ones $[\cdot]$. If α and β are arbitrary entities and a relation $\mathcal{R}$ exists among them, then the Iverson brackets $[\alpha \mathcal{R} \beta]$ evaluates to 1 otherwise the result is 0. In general the notation can be extended so that given a statement P:

$$[P] = \begin{cases} 1 \text{ if } P \text{ is true} \\ 0 \text{ if } P \text{ is false} \end{cases}$$

Cleary the Iverson brackets satisfy the following properties:

- $[P \wedge Q] = [P][Q]$
- $[P \vee Q] = [P] + [Q] - [P][Q]$ (equivalent to inclusion exclusion theorem)
- $[\neg P] = 1 - [P]$
- $[P \oplus Q] = \mid [P] - [Q] \mid$
- $[\forall m \ P(m)] = \prod_m [P(m)]$
- $[\exists m \ P(m)] = \min\{1, \sum_m [P(m)]\}$

The Iverson brackets can be also used to rewrite in a compact way the definition of certain functions like the Kronecker delta: $\delta_{jk} = [j = k]$.

- *Polyhedra*

In the metric space $(\mathbb{R}^n, \|\cdot\|_2)$ the following geometric entities can be defined. A **halfspace** is defined as the set $H = \{\boldsymbol{x} \in \mathbb{R}^n | \boldsymbol{a}^T\boldsymbol{x} \leq b\}$, where $\boldsymbol{a} \in \mathbb{R}^n, b \in \mathbb{R}$ A **polyhedron** is defined as the intersection of a finite set of halfspaces, i.e.:

$$P = \bigcap_i H_i = \bigcap_i \{\boldsymbol{x} \in \mathbb{R}^n \mid (\boldsymbol{a}^i)^T\boldsymbol{x} \leq b_i\} = \{\boldsymbol{x} \in \mathbb{R}^n | A\boldsymbol{x} \leq \boldsymbol{b}\}$$

where $A \triangleq \begin{bmatrix} (\boldsymbol{a}^1)^T \\ (\boldsymbol{a}^2)^T \\ \vdots \\ (\boldsymbol{a}^n)^T \end{bmatrix}$ and $\boldsymbol{b} \triangleq \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$.

A **polytope** is defined as a bounded polyhedron, i.e.:

$$P = \{\boldsymbol{x} \in \mathbb{R}^n | A\boldsymbol{x} \leq \boldsymbol{b} \ \wedge \ \|\boldsymbol{x}\|_2 < M\}, \text{ for some } M > 0$$

P is said to belong to the set of the polytopes on $\mathbb{R}^n$ denoted as $P \in \langle\mathbb{R}^n\rangle$. Polyhedra and polytopes are particularly interesting in the context of neural network verification as "fast" algorithms to compute the solution of linear programming and integer linear programming problems defined over such domains are known. Many state-of-the-art verifiers often exploit integer linear programming solvers as subroutines.

- *Topological Spaces & S^{n-1}*

The formalism introduced here follows [9]. Let X be a non empty set. A topological structure or simply topology on X is a non empty family $\mathcal{T}$ of subsets of X, that are the open sets of the topology. The subsets must follow these conditions:

A1) $\emptyset$ and X are open;

A2) The union of any family of open sets is an open set;

A3) The intersection of any two open sets is an open set.

A particularly interesting object in topology is S^{n-1} which is usually defined as:

$S^{n-1} \triangleq \{\boldsymbol{x} \in \mathbb{R}^n | \, \|\boldsymbol{x}\| = 1\}$

However in this thesis S^1 was defined in a slightly different manner, i.e. it is the result of the "gluing" of $-\pi$ and π from the interval $[-\pi, \pi]$

- *Mathematical Programming (MP) and Linear Programming (LP)*

A linear programming problem is a particular case of the general mathematical programming problem in which the objective function f is linear and the domain X is a polytope.

Mathematical programming problem:
$\max_{\boldsymbol{x} \in X \subseteq \mathbb{R}^n} f(\boldsymbol{x})$

Linear programming problem:

$$\begin{cases} \max\limits_{\boldsymbol{x} \in X} \boldsymbol{c}^T \boldsymbol{x} \\ X = \{\boldsymbol{x} \mid A\boldsymbol{x} \leq \boldsymbol{b} \wedge \boldsymbol{x} \geq \boldsymbol{0}\} \end{cases}$$

Acknowledgments

I am deeply grateful to prof. Armando Tacchella for the supervision of this thesis.

I would like to express my most sincere gratitude to prof. Alessandro Giusti and prof. Jérôme Guzzi for the help with the pulp-frontnet case study.

Special thanks go to dott. Stefano Demarchi for the supervision of the code developed for chapter 3, prof. Luca Oneto for some interesting aspects developed in this thesis for the supervision of the code developed for chapter 5 and for the material regarding machine learning.

I would also like to thank prof. Fabio Roli for the material on adversarial machine learning and the EU AI Act, prof. Sara Negri for the material concerning the relation between logic and NN verification, prof. Marcello Sanguineti for the material involving the relation between linear programming and NN verification, prof. Perla Maiolino for the material relative to soft robotics and machine learning.

Last but not least i would really like to thank dott. Giovanni Mottola and prof. Antonio Sgorbissa for sharing datasets [O] and [P].

References

1) **Evaluating Reachability Algorithms for Neural Networks on Never2**
Dario Guidotti, Stefano Demarchi, Luca Pulina, Armando Tacchella

1a) **pyNeVer: a Framework for Learning and Verification of Neural Networks**
Dario Guidotti, Luca Pulina, Armando Tacchella

1b) **Experimenting with Constraint Programming Techniques in Artificial Intelligence: Automated System Design and Verification of Neural Networks**
Stefano Demarchi (PhD Thesis)

2) **Fooling A Complete Neural Network Verifier**
Dániel Zombori, Balázs Bánhelyi, Tibor Csendes, István Megyeri, Márk Jelasity

3) **Exploiting Verified Neural Networks via Floating Point Numerical Error**
Kai Jia, Martin Rinard

4) **Introduction to Neural Network Verification**
Aws Albarghouthi

5) **Artificial Intelligence A modern approach (Third Tdition)**
Stuart Russel, Peter Norvig

6) **Fully Onboard AI-powered Human-Drone Pose Estimation on Ultra-low Power Autonomous Flying Nano-UAVs**
Daniele Palossi, Nicky Zimmerman, Alessio Burrello, Francesco Conti, Hanna Müller, Luca Maria Gambardella, Luca Benini, Alessandro Giusti, Jérôme Guzzi

7) **Adversarial Examples Are Not Easily Detected: Bypassing Ten Detection Method**
Nicholas Carlini, David Wagner

8) **Towards Evaluating the Robustness of Neural Networks**
Nicholas Carlini, David Wagner

9) **Geometria 2 (chapter 1: pag. 10, chapter 2: pag. 42 & 72-73)**
Edoardo Sernesi

10) **A programming language**
Kenneth E. Iverson

11) **Star-Based Reachability Analysis of Deep Neural Networks**
Hoang-Dung Tran, Diago Manzanas Lopez, Patrick Musau, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T. Johnson

12) **Explaining and Harnessing Adversarial Examples**
Ian J. Goodfellow, Jonathon Shlens, Christian Szegedy

13) **RobotGPT: Robot Manipulation Learning from ChatGPT**
Yixiang Jin, Dingzhe Li, Young A, Jun Shi, Peng Hao, Fuchun Sun, Jianwei Zhang, Bin Fang

14) **Case Study: Safety Verification of an Unmanned Underwater Vehicle**
Diego Manzanas Lopez, Patrick Musau, Nathaniel Hamilton, Hoang-Dung Tran, Taylor T. Jonhson

15) **Reinforcement Learning Approaches in Social Robotics**
Neziha Akalin, Amy Loutfi

16) **4D printing soft robots guided by machine learning and finite element models**
Ali Zolfagharian, Lorena Durran, Saleh Gharaie, Bernard Rolfe, Akif Kayanak, Mahdi Bodaghi

17) **Model-free Soft-Structure Reconstruction for Proprioception using Tactile Arrays**
Luca Scimeca, Josie Hughes, Perla Maiolino, Fumiya Iida

18) **Discovering Adversarial Driving Maneuvers against Autonomous Vehicles**
Ruoyu Song, Muslum Ozgur Ozmen, Hyungsub Kim, Raymon Muller, Z. Berkay Celik, Antonio Bianchi

19) **Classification with Quantum Machine Learning: A survey**
Zainab Abohashima, Mohamad Elhoseny, Essam H. Houssein, Waleed M. Mohamed

20) **Quantum Machine Learning**
Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, Seth Lloyd

21) **Quantum Associative Memory**
Dan Ventura, Tony Martinez

22) **Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm**
David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Lauren Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, Demis Hassabis

23) **A neural network designed to solve the N-Queens problem**
Jacek Mańdziuk and Bohdan Macukow

Links

A) https://futureoflife.org/project/eu-ai-act/

B) https://www.bitcraze.io/

C) https://www.bitcraze.io/documentation/system/platform/cf2-architecture/

D) https://github.com/idsia-robotics/pulp-frontnet/tree/main

E) https://docs.python.org/3.8/library/pickle.html

F) https://www.bitcraze.io/documentation/repository/aideck-gap8-examples/master/infrastructure/gap8/

G) https://mpmath.org/doc/current/mpmath.pdf

H) https://mpmath.org/doc/1.3.0/

I) https://oeis.org/A000170

J) https://fredrikj.net

K) https://oeis.org/A002485

L) https://oeis.org/A002486

M) https://github.com/FabrizioLeopardi/LF_Adversarial/

N) https://bostondynamics.com/blog/starting-on-the-right-foot-with-reinforcement-learning/

O) https://universe.roboflow.com/fabio-conti/solar-panles-instance-segmentation

P) https://universe.roboflow.com/raise/test_iros_2024_detection_containers_mottola_mar2024

Q) https://futureoflife.org/project/eu-ai-act/