# Projet Long : Sûreté de l'IA

**Baptiste COMBELLES**
**Clément CONTET**
**Guillaume COULAUD**
**Joceran GOUNEAU**
**Thibault MOUSSET**

Encadré par :
**Guillaume DUPONT**, **Aurélie HURAULT** & **Philippe QUÉINNEC**

Département Sciences du Numérique - Troisième année
2022-2023

# Contents

This report is accompanied by a `GitHub` repository: https://github.com/BaptisteCbl/surete-IA.

# 1 Introduction

Neural networks (NN) and especially Deep Neural Networks (DNN) have proven to be extremely powerful tools for machine learning tasks — such as recognition problems applied to images, text or speech. This work is focused on image classification for which NN has shown the ability to reach human-level accuracy. Nowadays, these models are commonly used for machine vision task such as recognizing road signs [1] which is crucial for self-driving cars. Thus, some guarantees are needed when using the prediction of a model in practical applications: will the model always recognize the "STOP" sign? can can it always have human-level accuracy or are there situations where its performance are worse?

Some works [2, 3] have shown that DNN have poor performance with specially created examples which are called adversarial examples. They can be defined as inputs specifically created by an attacker to deceive a classifier. In the case of image classification, the inputs are image which are modified to look like the original but for which the classification will not lead to the right result. However, a human should still be able to find the correct image's class.

There are defense mechanisms to counter such examples [4]. There are two main categories (1) Gradient Masking: as some attacks exploit the gradient information of the classifiers, masking or obfuscating the gradients aims to to make the attack less effective. (2) Robust Optimization : the goal is to train a robust classifier that can correctly classify the adversarial examples.

## 1.1 Model studied

### 1.1.1 Neural networks

We limit ourselves to deep feedforward networks [5] — or multilayer perceptrons. The goal of a feedforward network is to approximate a function $\hat{f}$. In the case of image classification $y = \hat{f}(x)$ with $\hat{f}$ the classifier, $x$ an image and $y$ the class or label of the image. A feedforward network aims to define a function $y = f(x, \theta)$ and to learn the parameters $\theta$ in order to provide the best result according to a specific criterion. The learning is made using backward propagation $\dfrac{\partial f(x,\theta)}{\partial \theta}$ and a gradient descent-like algorithm. The two feedforward network architecture studied are

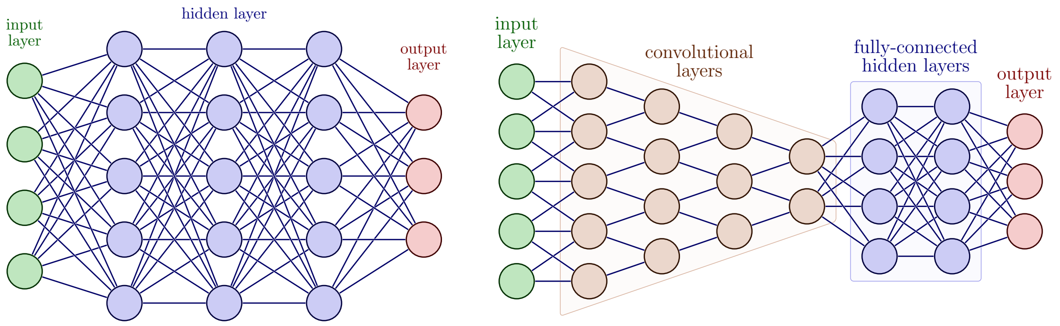
- Fully-Connected Neural Networks
  Fully-Connected neural networks (FCN) consists in a sequence of fully connected layers that connect every neuron in one layer to every neuron in the next layer. An

m-layer fully connected neural network can be formed as:

$$\begin{cases} z^0 = x \\ z^{\ell+1} = \sigma\left(W^{\ell} z^{\ell} + b^{\ell}\right). \end{cases}$$

The layer $\ell + 1$ consist of applying the activation function $\sigma$ to the product of the weight matrix $W^{\ell}$ and the previous layer $z^{\ell}$ plus a bias $b^{\ell}$

- Convolutional Neural Networks (CNN)
  Convolutional neural networks are used to process grid-like topology using convolution operation. They are therefore particularly suitable for images. The convolutional network can be seen as a simple feedforward network using a convolution in place of the matrix multiplication $W^{\ell} \cdot z^{\ell}$. The goal of a CNN is to extract the features of the object in the image in order to learn its representation.



(a) A fully connected neural network

(b) A convolutional neural network

Figure 1: Representation of the two models from https://tikz.net/neural_networks/

### 1.1.2 Decision trees

Decision trees are nodes organized in a tree structure in order to perform classification or regression. In addition to simple decision trees, two major models exist in the scientific literature about formal methods :

- decision tree ensembles (mostly random forest) which are sets of decision trees more or less independently trained where the final output is a combination of the outputs of each individual tree, often some kind of vote system;
- gradient boosted trees: dynamically built ensemble where models are trained sequentially in order to slightly improve on past models' performances.

# 2 Bibliography

## 2.1 Formal verification

### 2.1.1 Goals

Among the multiple properties that can be verified on machine learning models[6], the one that interests us the most here is the robustness of input perturbation. This property focuses on the study of the impact on outputs of perturbations of the inputs (goal G4). Typically, such a property ensures that all points within a ball with a certain radius around a given input are classified in the same way as the original input. In the case of NNs, it is also possible to pursue a similar but slightly different goal ensuring that none of the outputs are hazardous, i.e. that a certain property on the outputs holds independently of the inputs (goal G5).

### 2.1.2 Methods

Similarly to what Urban and Miné did in their survey [7], we distinguish between two main classes of methods used in formal verification: complete methods and incomplete methods. If both classes of methods are sound, incomplete methods will not always be able to conclude whether or not a given property holds. This is due to the fact that incomplete methods do not solve the verification task directly, they rather solve a relaxation of the problem. This makes incomplete methods prone to false positives meaning that a warning will be triggered because the method can not verify some inputs due to overapproximation while in fact, those inputs are well-behaved. Nevertheless, incomplete methods are not to be discarded, indeed, their complete counterpart usually suffer from being much less scalable and having a restricted scope in terms of the specific choices of the verified model (type of architecture or activation function for NN for instance) while the use of approximations makes incomplete methods more universal. It is also worth noting that it is possible to make incomplete methods asymptotically complete by iteratively refining the underlying over-approximating analysis when it is inconclusive.

Verification of robustness for decision trees is quite similar to verification for neural networks as the purpose of it is often finding the nearest adversarial input by using similar formal tools.

In the following sections, we will present multiple complete formal approaches: satisfiability modulo theories based approaches (SMT) (Section 2.1.2.1), mixed integer linear programming-based approaches (MILP) (Section 2.1.2.2), and another approach (Section 2.1.2.4). Additionally, we will describe one incomplete method, static analysis by abstract interpretation (Section 2.1.2.3).

#### 2.1.2.1 SMT-based methods

In order to be able to apply SMT approaches, the model is first encoded as first-order logic formulae and its negation of the desired property is added to it. If the problem is satisfactory then we have a counterexample and the property is not verified. Otherwise, the problem is unsatisfiable, there exist no inputs making the formulae valid and hence, there is no counter-example and the property is verified. In the case of a feed-forward neural network, the problem can be encoded in the following way [8]:

$$\begin{aligned}
&\hat{z}^{\ell+1} = W^{\ell+1} z^{\ell} + b^{\ell+1} && \forall \ell \in [\![0, n-1]\!] && \text{(1a)}\\
&z^{\ell} = \max\{0, \hat{z}^{\ell}\} && \forall \ell \in [\![0, n-1]\!] && \text{(1b)}\\
&l \leqslant z^0 \leqslant u && && \text{(1c)}\\
&z^n \leqslant 0 && && \text{(1d)}
\end{aligned}$$

where Equation (1a) represents the affine transformations between two consecutive layers, Equation (1b) encodes the Relu activation functions, Equation (1c) describes the constraints on the inputs, and Equation (1d) is the negation of the property $z^n > 0$ on the output which is under scrutiny.
It is important to note that in the case of NNs, the non-linearity of the activation functions, a key element in their performance, is a serious drawback since it makes the problem NP-complete.

#### 2.1.2.2 MILP-based methods

In this context, mixed integer linear programming (MILP) is used in a similar way to search for a potential counterexample to the verified property. To this end, objective functions often take the form of a minimum or a maximum. In MILP encoding, the difficulty due to the non-linearity of the activation functions also comes to light since the system can only manage linear equations. To get around the problem, binary variables are introduced to account for the activation status. In the same context as in the previous section, a possible MILP encoding presented by Tjeng et al. [9] is:

$$\begin{aligned}
&\hat{z}^{\ell+1} = W^{\ell+1} z^{\ell} + b^{\ell+1} && \forall \ell \in [\![0, n-1]\!] && \text{(2a)}\\
&\delta^{\ell} \in \{0,1\}^{|z^{\ell}|},\ 0 \leqslant z^{\ell} \leqslant u^{\ell} \cdot \delta^{\ell},\ \hat{z}^{\ell} \leqslant z^{\ell} \leqslant \hat{z}^{\ell} - l^{\ell} \cdot (1 - \delta^{\ell}) && \forall i \in [\![0, n-1]\!] && \text{(2b)}\\
&l \leqslant z^0 \leqslant u && && \text{(2c)}\\
&\min\ z^n && && \text{(2d)}
\end{aligned}$$

where equations (2a) and (2c) play the same role as their respective counterparts (1a) and (1c), equation (2b) is the translation of equation (1b) in linear terms using binary variables and (2d) is the objective function, in this case, if the minimum obtained is negative, the property $x^n > 0$ will not be verified.

#### 2.1.2.3 Abstract Interpretation-based methods

Abstract Interpretation is a classical approach used in static analysis to derive important properties. In our case, the method is used to prove the local robustness of a neural network. An abstract domain is chosen to abstract the computation of the model. Multiple domains exist (intervals, zonotopes, ...) supporting different activation functions. However, having complex domains can make the verification harder. Urban et al. give a more detailed presentation of the method in the context of machine learning [7]. Abstract Interpretation has also been used to detect bias [10]. It uses forward analysis on simple domains to group the input space regarding activation functions and then perform backward analysis to discriminate biased partitions.

#### 2.1.2.4 Other approaches

A method was proposed in 2019 by Hongge Chen et al. [11] to exactly verify the robustness of decision trees (ensemble trees) by computing the minimal adversarial distortion. The algorithm developed allows to find it in a linear time for a single tree. For ensemble trees, the problem is changed to a max-clique enumeration with a multi-level algorithm for large models.

### 2.1.3 State of the art

Formal verification of machine learning models is a very fast-evolving field. The size of models that can be verified while lacking behind the size of the models currently developed is still growing fast. This makes it hard to keep track of what the state of the art looks like. Nevertheless, urban et al. give a good overview of what was actually achievable two years ago [7].

Additionally, a competition of verification has been held in 2022 [12] to rank the latest verification tools. Hence it gives us a good idea of the performance of the bleeding-edge technology in this area. The hardware used in the competition is the same for all tools (an AWS solution). Multiple types of networks have been tested: complex UNet, convolutive, ResNet, and other quite simple networks. They usually have from a few hundred to hundreds of thousands of neurons with an input dimension of approximately 1000. Such technology can verify up to a thousand networks in 5 minutes for the best ones. As an example, the winner of the competition is a tool called $\alpha, \beta - CROWN$ which uses GPU-accelerated bound propagation algorithm. For more details of the competition please see the original paper.

## 2.2 Adversarial attacks

### 2.2.1 Neural Networks

Adversarial examples are the result of models being too linear [3].

The attacker's goal is to generate a fake image $x'$, called the adversarial example, such that $x'$ is similar to $x$ to the human eye but misleads the classifier. In other words,

$$\begin{aligned}&\text{find } x' \text{ satisfying } \|x' - x\| \leqslant \varepsilon\\&\text{such that } C(x') = t \neq y\end{aligned}$$

We call adaptive adversaries, the attacks that are tailored to the particular details of the model's defense and attempt to invalidate the assertions of robustness. The different attacks and defenses tactics are described in the papers [13, 4, 14].

#### 2.2.1.1 White-box attacks

In a white-box attack, the attacker has access to the classifier $C$, or more generally the model $F$ and the victim sample $(x, y)$. For instance, $x$ can be an image and $y$ a label. We now list a few methods to generate $x'$.

**Szegedy's L-BFGS Attack** The problem is formulated as a search for the closest adversarial example $x'$ to $x$, it is a targeted attack

$$\begin{aligned}&\text{minimize } \|x - x'\|_2^2 \\ &\text{subject to } C(x') = t \text{ and } x' \in [0,1]^m\end{aligned} \tag{1}$$

The term $x' \in [0,1]^m$ is called the box constraint, it aims to guarantee that $x'$ will be within the definition domain of the images.

The problem is reformulated by introducing the loss function $\mathcal{L}$ to penalize the function to favor the classification of $x'$ as $t$. Finally, the parameter $c$ is involved in the search for the minimal distance to $x$.

$$\begin{aligned}&\text{minimize } c\|x - x'\|_2^2 + \mathcal{L}(\theta, x', t) \\ &\text{subject to } x' \in [O,1]^m\end{aligned} \tag{2}$$

This optimization problem is solved with the L-BFGS algorithm.

**Fast Gradient Sign Method (FGSM)** It is a one-step method that can lead to a targeted or untargeted attack. The formulation is:

$$x' = x + \varepsilon\,\text{sign}(\nabla_x \mathcal{L}(\theta, x, y)) \text{ non-target} \tag{3}$$

$$x' = x - \varepsilon\,\text{sign}(\nabla_x \mathcal{L}(\theta, x, t)) \text{ target t} \tag{4}$$

For a targeted attack Equation 4 can be seen as a one-step gradient descent to solve the following problem.

$$\begin{aligned}&\text{minimize } \mathcal{L}(\theta, x', t) \\ &\text{subject to } \|x' - x\|_\infty \leqslant \varepsilon \text{ and } x' \in [0,1]^m\end{aligned} \tag{5}$$

The benefits of the attack are its ease of use and the fact it is fast to generate an example.

**Basic Iterative Method (BIM).** Basic Iterative Method can be seen as an iterative version of the FGSM attack, it can also be a targeted or untargeted attack, the untargeted formulation is:

$$\begin{aligned} &x_0 = x \\ &x^{t+1} = \Pi_{x,\varepsilon}\left[x^t + \alpha\,\mathrm{sign}(\nabla_x \mathcal{L}\theta, x^t, y))\right] \end{aligned} \tag{6}$$

With $\Pi_{x,\varepsilon}$ being the projection on the $\varepsilon$-neighbor ball of $x$ and $\alpha$ the step size.

A variation of this method is the Projected Gradient Descent (PGD) [14], which has an added random initialization step of $x_0$ within $x \pm \varepsilon$.

**Jacobian-Based Saliency Map Attack (JSMA).** We compute the Jacobian of the score function $F$. It can be seen as a greedy attack algorithm by iteratively manipulating the pixel which is the most influential to the model output. We use

$$J_F(x) = \frac{\partial F(x)}{\partial x} = \left\{\frac{\partial F_j(x)}{\partial_i x}\right\}_{i\times j} \tag{7}$$

to model $F(x)$'s change in response to changes of its input $x$. Then, we build an adversarial saliency map indicating which input features an adversary should perturb to obtain the desired changes in the model output.

The term saliency map comes from computer vision. It is an image that highlights the region on which people's eyes focus first.

**Carlini & Wagner's Attack** It is a reformulation of the targeted L-BFGS problem:

$$\begin{aligned} &\text{minimize } \|x - x'\|_2^2 + c \cdot f(x', t) \\ &\text{subject to } x' \in [0,1]^m \end{aligned} \tag{8}$$

$f$ is defined as $f(x', t) = \max_{i \neq t} Z(x')_i - Z(x')_t . i \neq t$. With $Z$ such that $F(x) = \mathrm{softmax}\,(Z(x)) = y$

Minimizing $f$ encourages the algorithm to find an $x'$ that has a larger score for class $t$. By applying line search on constant $c$ we can find an $x'$ that has the least distance to $x$.

$f$ is called the margin loss function.

**Deep Fool** It is an untargeted attack. It assumes that the NN is totally linear and hyperplanes separate the class. Then the minimal perturbation is given by the projection from one space to another.

#### 2.2.1.2 Black-box attacks

A black box attack is a type of adversarial attack in which the attacker does not have access to the internal workings of the AI system they are attacking. The attacker can only manipulate the inputs of the network and see the output.

**Substitute model** The goal of this attack is to exploit "transferability": a sample $x'$ that can attack $F_1$ is also likely to attack $F_2$. The main steps of the attack are:

1. Synthesise the Substitute Training Dataset
2. Train the Substitute model
3. Augment the Dataset
4. Attack the substitute model

**Model Inversion / Extraction** The attacker inputs known data into the black box model and examines the model's output. By analyzing the output, the attacker can learn information about the model's internal workings, including its training data, architecture, and parameters.

**Zeroth Order Optimization (ZOO)** It needs to have access to the confidence score associated to the output label. It enables the gathering of information on the gradient around the sample $x$ by observing the changes in the confidence score as the image is perturbed. We can approximate the gradient for a given index $i$ with the following formulation:

$$\frac{\partial F(x)}{\partial x_i} \approx \frac{F(x + he_i) - F(x - he_i)}{2h}$$

We can now use a gradient-based attack. The success rate of ZOO is higher than with a substitute model.

**ColorFool** [15] This method is specific to images; it uses a segmentation network to decompose an image into semantic regions and then simply modify the $a$ and $b$ values in the *Lab* space for each of these regions. In order to keep images that still seem natural, these color modifications lie within a specific restrained range for regions that are sensitive to the human eye (person, sky, vegetation, and water); for other regions these modifications are unrestrained. For a given image and a given model, ColorFool gets multiple attempts at fooling the model, with modifications of increasing magnitude and the output of the model as only feedback. This method has shown results comparable to white-box attacks, is robust to denoising, and has moreover shown improved transferability (which is the success rate of misleading an unseen classifier).

**Hop Skip Jump Attack (HSJA)** This attack [16] aims to estimate the gradient using binary information at the decision boundary

**SPSA** The goal is to use the Simultaneous Perturbation Stochastic Approximation[1] (SPSA)

[1] SPSA: https://www.jhuapl.edu/spsa/

algorithm to estimate the gradient [17].

#### 2.2.1.3 Poisoning attacks

The concept of this attack is to change the training data set in order to manipulate its behavior. It is quite difficult to detect when datasets are huge.

We can poison the model in different ways :

**Label Flipping** The attacker modifies the labels of the training data, causing the AI system to learn incorrect relationships between inputs and outputs.

**Data Injection** The attacker adds malicious data to the training data, causing the AI system to learn incorrect relationships between inputs and outputs.

**Model Parameter Modification** The attacker modifies the parameters of the AI model, causing it to make incorrect predictions.

**Koh's Model Explanation** It is a method to interpret DNNs. A model can explicitly quantify the change in the final loss without retraining the model when only one training is modified. This work can be adapted to poisoning attacks by finding those training samples that have a large influence on the model's prediction.

**Poison Frogs** Insert an adversarial image with a true label to the training set in order to cause the trained model to wrongly classify a targeted sample.

#### 2.2.1.4 Grey-box attacks

The attacker trains a Generative adversarial network (GAN) [18] targeting the model of interest [19]. A GAN is a model in which two neural networks, the generator, and the discriminator, compete with each other to become more accurate in their predictions. The goal is to achieve high-fidelity generation.

The attacker can craft adversarial examples from the GAN. The main benefit is the acceleration of the process to produce examples once the model is trained. This allows to shift the cost of creating adversarial examples, as the method implies a big initial cost with the training but afterward the cost to produce an example is cheap. This is particularly interesting for adversarial training.

### 2.2.2 Decision Trees

Unlike Neural Networks, the algorithm to generate adversarial examples is poorly studied for tree-based models. The main reason is that the method relying on gradient cant be used as the models are discrete and non-differentiable.

**Greedy search** A first approach described in [20] is based on greedy search. It consists in searching the neighborhood of the leaf producing the original prediction to find another leaf labeled as a different class, it is the adversarial leaf. Then, it is necessary to find the path from the original leaf to the adversarial leaf. Finally, the sample is modified in relation to the conditions on the path from the original leaf to the adversarial leaf. Thus, forcing the decision tree to misclassify the sample.

**Cheng's attack** It focuses on the distance between the benign example and the decision boundary. The problem of finding an adversarial example is reformulated as a minimization of this distance. It exploits the fact that the distance to the decision boundary is usually smooth within a local region and can be found by binary search given a direction vector

**Kantchelian attack.** It is a method relying on MILP to find the exact smallest distortion necessary to mislead the model.

## 2.3 Building Safer Models

### 2.3.1 Countermeasures Against Adversarial Examples

There are three types of countermeasures 1) Gradient Masking/Obfuscation, 2) Robust Optimization, and 3) Adversarial examples detection.

#### 2.3.1.1 Gradient Masking/Obfuscation

**Defensive distillation:** "Distillation" is a technique to reduce the size of DNN architectures. We define the temperature $T$ of a softmax for a vector $x$ of size $K$ as the following:

$$\text{softmax}(x, T)_i = \frac{e^{x_i/T}}{\sum_{j=1}^{K} e^{x_j/T}} \quad for\ i = 1, 2, \dots, K$$

An example of a training procedure with distillation is:

1. Train a network $F$ on the given training set $(X, Y)$ by setting the temperature of the softmax to $T$.
2. Compute the scores given by $F(X)$ again and evaluate the scores at temperature $T$
3. Train another network $F'_T$, the *distilled* model, using softmax at temperature $T$ on the dataset with soft labels $(X, F(X))$.
4. Use the distilled network with softmax at temperature $T = 1$, which is denoted $F'_1$ during prediction on test data (or adversarial example)

With the softmax we cause the inputs to become larger by a factor of $T$, meaning for a sample $x$ its neighbor $x'$ will be $T$ times larger.

**Shattered Gradients:** The goal is to protect the model by prepossessing the data. We use a non-smooth or non-differentiable pre-processor $g(.)$ and then train a DNN model $f$ on $g(X)$. The trained classifier on $f(g(X))$ is not differentiable in terms of $x$, causing the failure of adversarial attacks. thermometer encoding uses a preprocessor to discretize an image's pixel value $x_i$ into a $l$ -dimensional vector.

**Stochastic/Randomized Gradients:** The goal is to randomize the DNN model in order to confound the adversary. We train a set of classifiers $s = F_t : t = 1, 2, ..., k$. During the evaluation of data $x$, we randomly select one classifier from the set $s$ and predict the label $y$. As the adversary has no idea which classifier is used by the prediction model, the attack success rate will be reduced. Dropout can also be added.

**Exploding & Vanishing Gradients:** We use a generative model to project a potential adversarial example onto the benign data manifold before classifying them. Adding a generative model before the classifier DNN will cause the final classification model to be extremely deep. The cumulative product of partial derivatives from each layer will cause the gradient to be extremely small or irregularly large, which prevents the attacker from accurately estimating the location of adversarial examples.

#### 2.3.1.2 Adversarial Training

Robust training aims to improve the classifier's robustness by changing the manner of learning. Major focus:

1. Minimize the average adversarial loss
2. Maximize the average minimal perturbation distance

Typically, a robust optimization algorithm should have prior knowledge of its potential threat or potential attack. The defenders build classifiers that are safe against this specific attack.

**Adversarial training with FGSM.** We use Non-targeted FGSM to generate adversarial example $x'$ for the training dataset. Then we add the adversarial example and its true lab $(x', y)$ into the training dataset. The goal is to increase the model's robustness against FGSM attacks. But it is still vulnerable to iterative attacks.

This approach can be improved for scalability with the use of batch and batch normalization.

**Adversarial training with PGD** [14] The Projected gradient descent attack can be seen as a heuristic method to find the "most adversarial" example in the $l_\infty$ ball around $x$. The training on the most-adversarial examples solves the problem of learning model parameters $\theta$ that minimize the adversarial loss.

The model is trained only on adversarial examples, instead of a mix of normal samples and adversarial examples. It gives good robustness against both single-step and iterative attacks on MNIST and CIFAR10. However, the training is hard to scale as the method involves an iterative attack to generate all training samples.

**Ensemble adversarial training.** The goal is to protect the model against the single-step attack and can be applied to large datasets such as ImageNet.

It consists in augmenting the classifier's, $F$, the training set with adversarial example crafted from other pre-trained classifiers $F_1, F_2, \ldots$. Then, for each sample $x$, we use a single-step attack FGSM to craft adversarial examples $x'$ on the other classifiers. Because of the transferability of the single-step attack across the different models, $x'$ is also likely to mislead the classifier $F$. It means that these samples are a good approximation for the "most adversarial" example for model $F$ on $x$. Training these samples together will approximately minimize the adversarial loss.

**Accelerate adversarial training.** It is a "free" adversarial training that improves efficiency by reusing the backward pass calculation. The gradient of the loss to input: $\dfrac{\partial \mathcal{L}(x+\delta,\theta)}{\partial x}$ and the gradient of the loss to model parameters: $\dfrac{\partial \mathcal{L}(x+\delta,\theta)}{\partial \theta}$ can be computed together in one back propagation. (You Only Propagate Once (YOPO)). Thus accelerating adversarial training.

**Parametric Noise Injection** [21] It consists of injecting noise to an ensemble of vectors $\mathbf{v_k}$, $k \in [1, K]$, where each $\mathbf{v_k}$ can be the input, a layer's weights, or an inter-layer tensor of the model $f$. The added noise is defined as follows:

$$\bar{v_k^i} = v_k^i + \alpha_k \cdot \eta, \qquad \eta \sim \mathcal{N}(0, \sigma^2); \qquad \forall i \in [1, N_k] \tag{9}$$

$N_k$ being the length of the vector $\mathbf{v_k}$ and $\sigma = \sqrt{\dfrac{1}{N_k}\sum_i (v_k^i - \mu)^2}$ is the standard deviation of $\mathbf{v_k}$. $\alpha_k$ is a trainable parameter that is global to each vector $\mathbf{v_k}$, it is optimized during adversarial training on PGD attacks, with an added term in the loss that favors models whose predictions stay the same on adversarial examples in order to prevent these $\alpha_i$ to drop to 0.

**Feature Denoising** Most of the white-box gradient-based adversarial attacks result in adding a restricted noise to the input, so denoising the input could be a natural countermeasure to these attacks; the method developed in [22] takes this idea further and implements denoising features operation within the neural network.

**Fast adversarial training** Two papers offer an extension of the "free" adversarial training [23, 24]. In the same way as the ”free" training the two approaches exploit the backward propagation used to find the adversarial example to update the model. However to improve the convergence they use cyclical learning rate [25] — the learning rate is the coefficient to use in the gradient descent. In a basic training loop the learning rate is constant, and a bad value of this parameter leads to bad convergence, however, to find the correct value some hyperparameter tuning must be done which can be costly. The cyclical learning rate makes it possible not to depend on this tuning.

Furthermore, to improve the computing efficiency of the training, the implementation is made using mixed precision.

Catastrophic overfitting when using adversarial training with FGSM is also introduced in [23]. Catastrophic overfitting the accuracy of the model's evaluation suddenly decreases from an epoch, going from good to poor accuracy and showing no improvement for the following epochs.

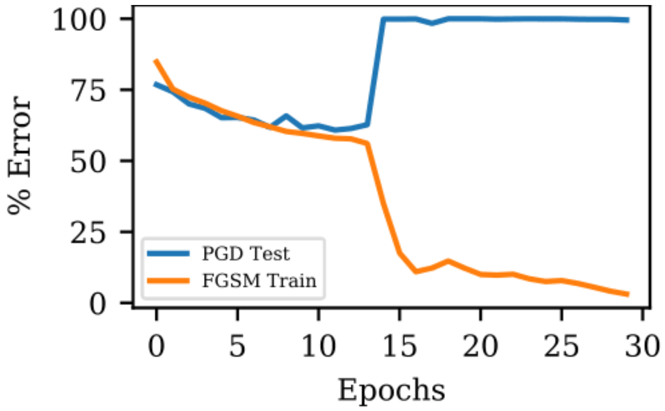


Figure 2: Illustration of catastrophic overfitting using FGSM adversarial training from https://github.com/locuslab/fast_adversarial

### 2.3.2 Confidence Learning

Confidence learning consists, in the case of a classification problem, to predict, as well as the class distribution, the overall confidence of the network in this distribution; as shown in the work of Chuan Guo et al. [26], modern neural networks have a tendency to be less calibrated, thus questioning the probability distribution interpretation of a classifier's output. Similarly to uncertainty prediction in regression tasks as done by Pavel Gurevich and Hannes Stuke [27]. Terrance DeVries et al. [28] proposed a method to address this problem, by dividing the output of a network $f_\theta$ into two branches: the usual class distribution output $p$ over $M$ classes, and a scalar value representing the confidence of the network $c$:

$$p, c = f_\theta(x) \qquad p_i, c \in [0,1], \sum_{i=1}^{M} p_i = 1 \tag{10}$$

During training, the predictions are modified using the confidence prediction to interpolate between the model's class prediction and the true label $y_i$:

$$\forall i \in [1, M], \qquad p'_i = c \cdot p_i + (1-c) y_i \tag{11}$$

These new predictions are used to compute the usual classification loss $\mathcal{L}_t$. A confidence loss is also computed so that the network always aims for $c = 1$:

$$\mathcal{L}_c = -\log c \tag{12}$$

The final loss we optimize for is then :

$$\mathcal{L} = \mathcal{L}_t + \lambda \mathcal{L}_c \tag{13}$$

$\lambda$ being a hyperparameter weighting the importance of the confidence loss.

### 2.3.3 Data preparation

**Data Sanitization.** It is the process of cleaning, validating, and transforming to remove or correcting errors, outliers, and other anomalies that could affect the accuracy and robustness of an AI system. It is useful when you can't be sure that the training is not poisoned.

**Data Augmentation.** It consists in adding additional training data which is generated from existing data by applying transformations such as rotations, translations, and rescalings.

### 2.3.4 Ensemble learning

Ensembling is a technique in which multiple AI models are combined to make predictions, with the final prediction being based on a combination of the predictions made by each of the models. This can help to improve the accuracy and robustness of the AI system, by reducing the risk of errors or biases in a single model [29].

### 2.3.5 Building easier to verify NNs

Since verifying the robustness of a model is a difficult task and is not always feasible another idea is to train them in a way that brings them back within the tractable problems category. The work of Baader et al. [30] proved that: for any continuous function $f$, there exists a feed-forward fully-connected neural network with ReLU activations whose abstract interpretation using the domains of intervals from an input region $B$ is an arbitrarily close approximation of $f$ on $B$. This implies that in some capacity, for any neural network, there exists another arbitrarily close neural network that can be more easily verified. This result was later extended by Wang et al. in [31] to the broader family of activation functions. Additionally, they studied the computational complexity of constructing neural networks obeying the criterion needed in their constructive proof and showed that this problem was strictly harder than NP-complete demonstrating that there was no universal efficient way of building an approximated model easier to verify.

Nevertheless, Xiao et al. [32], empirically highlighted two useful properties in order to obtain NNs easier to verify:

- Weight sparsity: having more zeros reduces the number of variables verifiers have to handle

- Activation stability: ensuring that given an input $x$ and a set of allowed perturbation $\mathrm{Adv}(x)$, the activation function stays in the same state over $\mathrm{Adv}(x)$. This enables the verifier to know in advance in which state the activation function will be thus preventing him from having to consider both branches

To achieve these goal they use a number of techniques:

- for weight sparsity:
  - $\ell_1$-regularization: a technique deterring the NN from relying too much on a subset of neurons by adding a term in the loss function proportional to the sum of the biases in absolute value;
  - small weight pruning: a post-process method setting weights under a set threshold to zero.
- Activation stability:
  - RS Loss: a regularization technique for inducing ReLU Stability by adding a term to the loss function based on estimates of the lower and upper bounds of each neurons pre-activation. This estimates is computed using interval arithmetic;
  - ReLU pruning: a post-process method replacing ReLU function by the null function or the identity if the ReLU is active or inactive on a large majority of inputs.

## 2.4 Explainability

Explainability is a rather difficult concept to define because of its elusive nature. Indeed, there is no clear definition of what an explication is. Marques-Silva and Ignatiev [33] draw the distinction between two different types of explications, the formal ones and the non-formal ones. Formal explainability is based on rigorously defined explications, it is therefore sound. In contrast, non-formal explainability is more based on a straightforward computation of a simpler and easier-to-understand model or on a set of features that justify the explanation without drawing too much attention to formal logic. This in return ensures that those explications are easier to understand at a human level while dropping the soundness. It is important to note that, in the field of explainability as in the whole field of machine learning, formal approaches are currently less popular. In the following sections, we give an example of each type of explication.

### 2.4.1 Minimal explication - formal explainability

Marques-Silva presents two types of explications based on the computation of minimal conjunctions of literals relating a feature with a specific range of values. He distinguishes two types of such explications [34].

**Abductive Explications** : informally, this type of explication is the minimal answer to the question "Why". It is a set of features with an assigned range of values such that if an input verifies the given formula then the prediction is guaranteed to be in a certain class, independently of the values assigned to the remaining features.

**Contrastive Explications** : in contrast this explication answers the question "Why not?". It gives a subset of features which, if allowed to take some other value, and when the remaining features remain unchanged, ensures that the prediction changes to a class other than the initial one.

This second type of explication enables us to clearly draw a parallel between explicability and verification since computing a contrastive explication is very similar to carrying out an adversarial attack on the model. This approach extracts features playing an important role in the decision of the model.

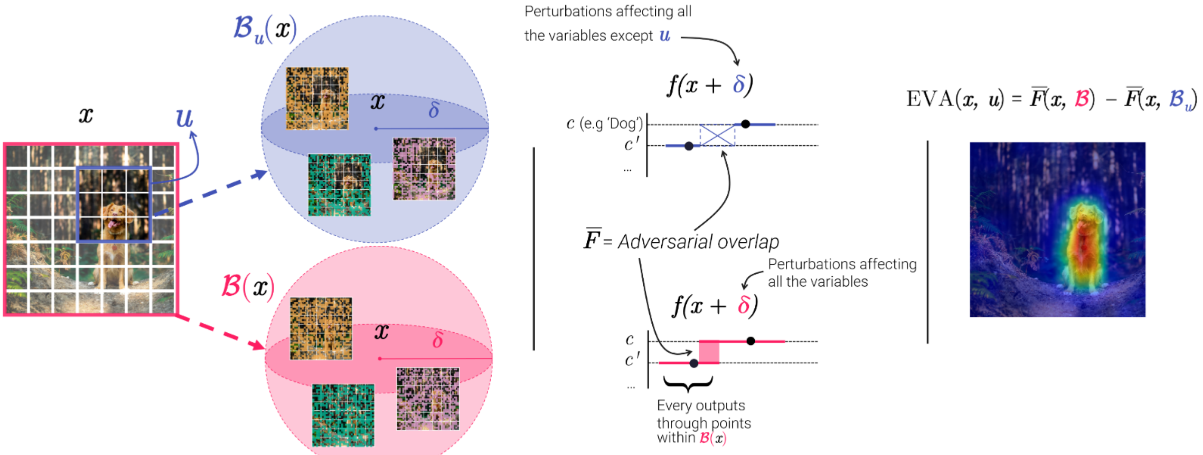
### 2.4.2 Verified perturbation analysis - non-formal explainability

According to Fel et al. [35], perturbations limited to the variables considered important should easily influence the model's decision. With that in mind, we can try to search for the most important variables of the model.

For a given input x and a bounded perturbation $\delta$, the verification methods allow us to obtain a minimum $f_{min}(x)$ and a maximum $f_{max}(x)$ bound on the output of a model. Formally $\forall\delta$ s.t $||\delta||_p \leqslant \epsilon$:

$$f_{min} \leqslant f(x+\delta) \leqslant f_{max}$$

This allows exploring the whole perturbation set without having to explicitly sample all the points.


$\mathcal{B}_u(x)$
$x$
$u$
$x$
$\delta$
$\mathcal{B}(x)$
$x$
$\delta$
Perturbations affecting all the variables except $u$
$f(x+\delta)$
$c$ (e.g 'Dog')
$c'$
...
$\overline{F}$ = Adversarial overlap
Perturbations affecting all the variables
$f(x+\delta)$
$c$
$c'$
...
Every outputs through points within $\mathcal{B}(x)$
$\text{EVA}(x, u) = \overline{F}(x, \mathcal{B}) - \overline{F}(x, \mathcal{B}_u)$

(1) $\mathcal{B}(x)$ are the points around $x$ of radius $\delta$. $\mathcal{B}_u(x)$ is a subset of $\mathcal{B}(x)$ where the variables $u$ are – the dog head – are not perturbed

(2) Without perturbating $u$, we can't reach any adversarial, thus $u$ is important

(3) Attribution after computing importance on each variables

Figure 3: Figure 2 from [35]

The EVA technique, short for Explainable Visual AI, is a method used to understand how neural networks make decisions about images.

To do this, EVA creates two versions of an input: one that is entirely perturbed and one that is partially perturbed. In the entirely perturbed version, the entire input is distorted, while in the partially perturbed version, only a portion of the input is distorted. By comparing the outputs of these two versions, EVA can determine which part of the input is more important for the decision.

Once the important parts of the input have been identified, EVA can then provide explanations for the neural network's decision. For example, if a neural network is classifying an image of a dog, EVA might identify that the presence of the dog's ears is crucial for the decision.

### 2.4.3 Explainable Neural Networks

[36] [37] [38] [34]

Another way to make more explainable models is to format their architecture. xNN Neural networks are an example of that.

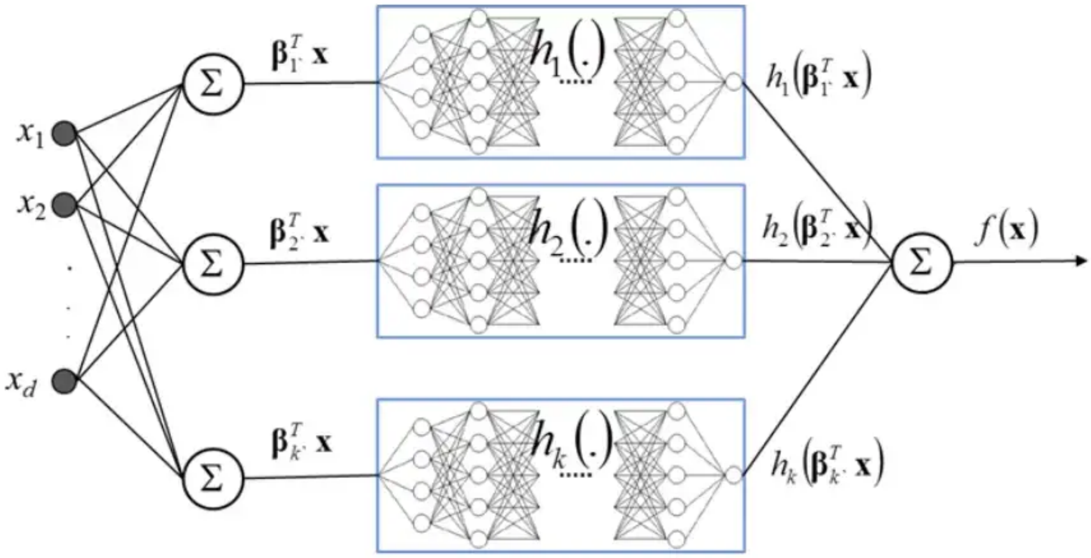


Figure 4: Architecture of a xNN Network

An xNN Network is composed of 3 elements :

1. The Projection Layer: It is fully connected to the input layer and each node in this layer learns a linear combination of the input features. The output of each node with a ridge function applied is used as the input to exactly one subnetwork.
2. Subnetworks: Their use is to enhance the ridge functions used in the projection layer. It's the element that gives us explanations of how the models learn.
3. The Combination Layer: It is composed of a single node that gives as output the weighted sum of all the ridge functions.

The reason we use the ridge function is that any continuous function can be approximated within an arbitrary precision by carefully selecting parameters in the network. On top of that, ridge functions are linear and do aren't affected by the dimensionality curse.

With this, we can get the scaled ridge functions and the projection coefficients of a model:

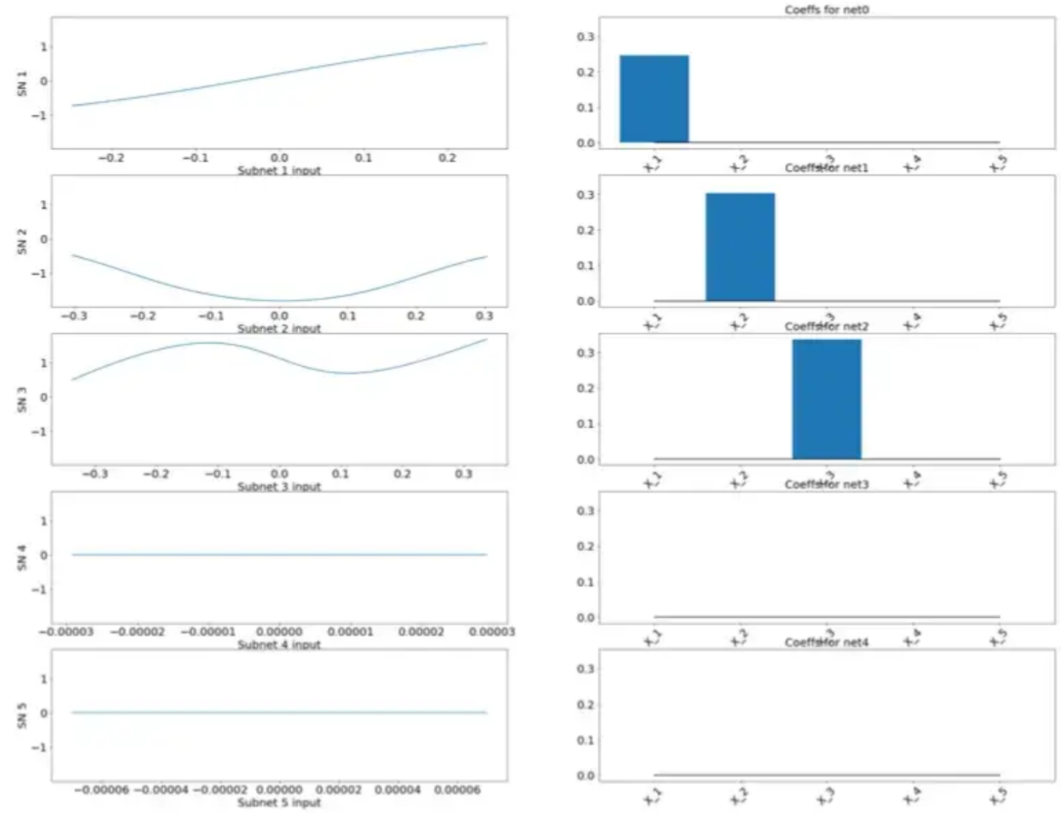


Figure 5: Results that can be created when using xNN Network

The projection coefficients reveal how the input features are combined to feed each of the ridge functions. When we talk about model recoverability, we mean the capacity to identify the original mechanisms that produced the data. On the other hand, explainability refers to the xNN's capability to clarify how it approximates a complicated function involving multiple variables, even if it doesn't accurately capture the underlying process that generated the data.

# 3 Plan for phase 2

## 3.1 Experiments

In light of the bibliographical study we carried out in phase 1, the objectives of this second phase are as follows:

1. Setting up the different technologies

2. Selecting and building vanilla and robustly trained models

3. Testing, analyzing and comparing models' performances on different points:
   - Verification work
   - Robustness to attacks
   - Trades-off in performance

### Point 1

In general, papers give access to the code they use. However, a part of them are using licensed software, paid cloud solution for hardware, or the code is not maintained and do not work anymore. In other cases, some pieces of code have uncommon libraries with little to no documentation.

### Point 2

This part is particularly important because there exist about as much different metrics and benchmarks as there are different papers. this makes it hard to compare results to have a broader view of the field. By having a hand on the whole process we are able to have a common reference frame to actually compare approaches.

Nevertheless, due to our limited resources and the issues discussed in point 1, we are currently not sure what we will be able to achieve.

### Point 3

Once we have been able to deploy the different technologies and have trained the models, we can benchmark and compare them. We plan to test the verification methods on three criteria: the size of models which can be verified, the time it takes, and whether or not it is able to prove the robustness of robust models.

Regarding the assessment of the robustness of a model the paper [39] gives a methodology to orientate the process and also gives some recommendations. The first steps in this evaluation are 1) Define a threat model: assumptions about the adversary's goals, knowledge, and capabilities. 2) Find the adaptive adversaries. Finally, we want to test how robust models compare to regular ones in terms of performance to evaluate the trade-off between robustness and accuracy.

## 3.2 Tools

There is a wide variety of tools used in the field and we do not know yet which we are going to use but this section presents a non-exhaustive list of the tools we encountered and plan to use.

### 3.2.1 Verification

**Neural Networks** To verify the robustness of neural networks we can use as references the paper [12] which reports the result of a competition of verification covering multiple formal methods mentioned above.

**Decision Trees** To verify the robustness of decision tree we use the code[2] associated to paper [11].

### 3.2.2 Attacks and defenses

**Neural Networks** To test the attacks and defenses (adversarial training) for neural networks we will study 3 libraries, DeepRobust, Cleverhans, and Foolbox [40, 41, 42]. Deeprobust proposes the implementation of the algorithms described in [4]. We also use the code[3] associated to paper [13].

**Decision trees** To test the attack and defense on decision trees, we use the code [4] associated to the paper [43].

### 3.2.3 Training easier to verify NNs

We plan to use the code[5] introduced in [32] to experiment with ReLU-stability impact on verification on models.

# 4 Experiments

Experiments We conduced multiple experiments among others:

1. Train the same model with and without adversarial training.
2. Compare the efficiency of attacks according to the training mode of the model.
3. Use the different verification methods on these models.

## 4.1 The framework

### 4.1.1 The datasets

The papers usually work with MNIST and CIFAR10. However, these datasets have some drawbacks: MNIST is an overused dataset which is too easy and does not represent modern classification problems. CIFAR10 is harder leading to the need to use relatively heavy

[2] treeVerification: https://github.com/chenhongge/treeVerification

[3] nn_robust_attack: https://github.com/carlini/nn_robust_attacks

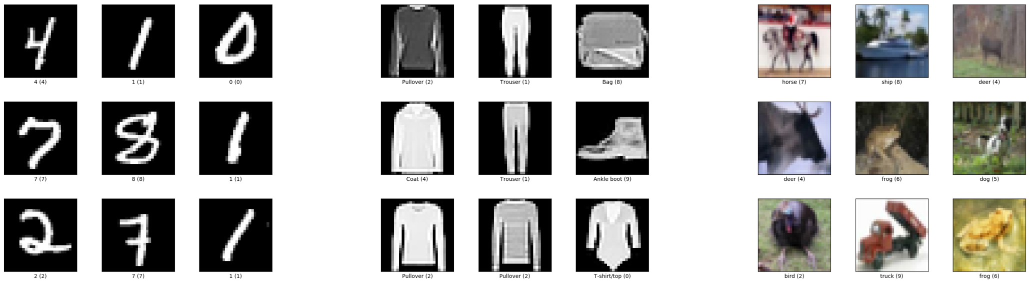
[4] RobustTrees: https://github.com/chenhongge/RobustTrees

[5] ReLU stability: https://github.com/MadryLab/relu_stable

models, which is not possible with our computing means. This is why we use Fashion-MNIST whose difficulty lies between MNIST and CIFAR10. The Table 1 describes some specific features of the three datasets and the Figure 6 shows some images from the dataset.

| | MNIST | FashionMNIST | CIFAR10 |
|---|---|---|---|
| Training samples | 60,000 | 60,000 | 50,000 |
| Test samples | 10,000 | 10,000 | 10,000 |
| Dimension | 28 × 28 | 28× 28 | 32× 32 |
| Color | Grayscale | Grayscale | RGB |

Table 1: Deatails on the datasets MNIST, FashionMNIST and CIFAR10



(a) MNIST (b) FashionMNIST (c) CIFAR10

Figure 6: Images from the three datasets, MNIST, FashionMNIST, and CIFAR10, *https://www.tensorflow.org/datasets/catalog/*

### 4.1.2 The model

As a model, we used the "small" CNN as in [32] it has 2 convolution layers and 2 fully connected layers which correspond to around 166.000 trainable parameters. The full model representation can be seen in Figure 13.

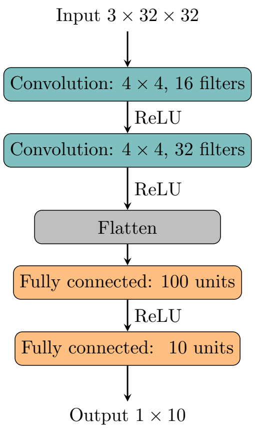

Input $3 \times 32 \times 32$
Convolution: $4 \times 4$, 16 filters
ReLU
Convolution: $4 \times 4$, 32 filters
ReLU
Flatten
Fully connected: 100 units
ReLU
Fully connected: 10 units
Output $1 \times 10$

Figure 7: The “small” CNN representation

### 4.1.3 Training a model

We trained the model previously described on the three datasets with and without adversarial training. We also benefit from the advantages of fast adversarial training to compute the accuracy of the evaluation datasets for all epochs. The code of the fast adversarial training is made in PytTorch and uses the NVIDIA apex library [6] to implement the mixed precision, however, the mixed precision part of this library is deprecated, and it is recommended to use the mixed precision implemented in PytTorch (PytTorch AMP) [7]. Furthermore, the code introduces in [23] provides a value for the cyclic learning rate that does not work with our model, which is not the case for the code of [24] which provides a value for a small CNN. For this reason, we will only use the second code.

The training is made with the basic training loop and with the fast approach. To perform the adversarial attacks we replaced all original images with adversarial examples computed with either FGSM or PGD attacks.

### 4.1.4 The attacks

The CleverHans library implements attacks in PyTorch and tensorflow, as we wanted the environment to be small enough to run on the computer at ENSEEIHT we added the used attacks and dependencies directly in the project. We now list the available attacks in PyTorch that can be easily added to an existing framework

- From CleverHans:
  - Fast Gradient Sign Method (FGSM)
  - Projected Gradient Descent (PGD)
  - Carlini & Wagner $l_2$ (CW)

[6] NVIDIA Apex: https://github.com/NVIDIA/apex
[7] PytTorch AMP: https://pytorch.org/docs/stable/amp.html

  - SPSA
  - Hop Skip Jump Attack (HSJA)

- From DeepRobust:
  - DeepFool

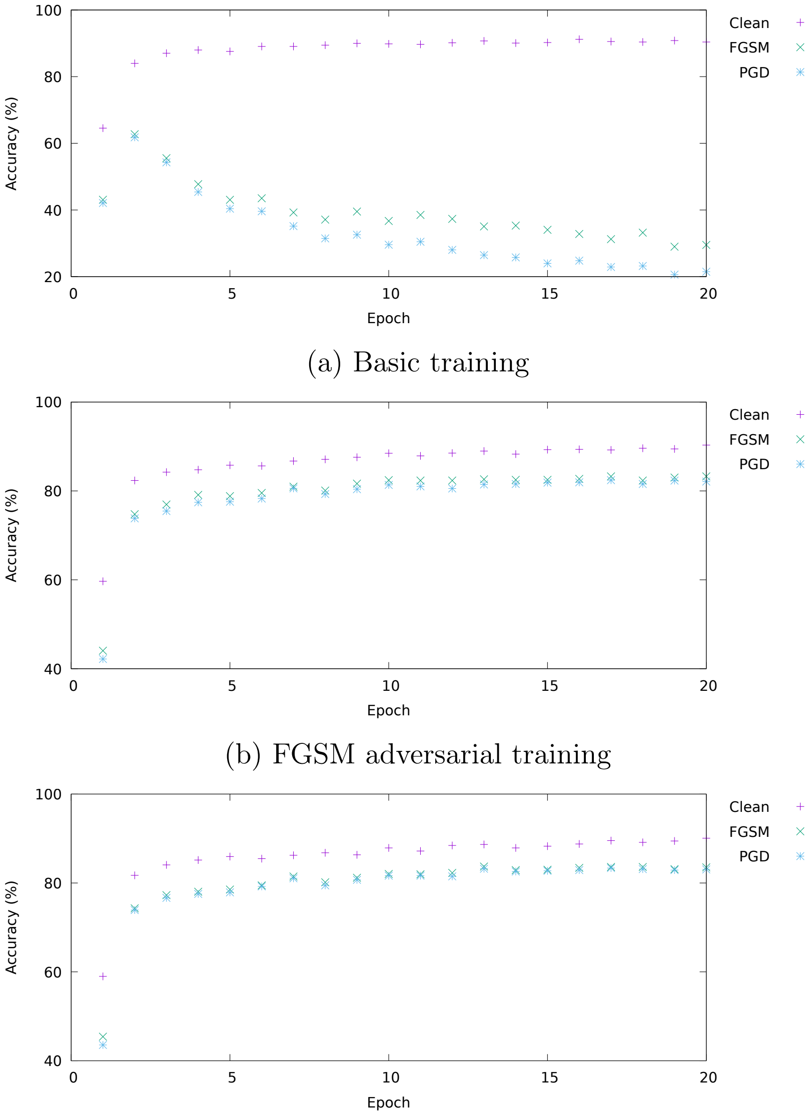
The available attacks can be classified as follows in Table 2, they can be separated into two classes gradient-based and non-gradient based with several available norms.

| Features | FGSM | PGD | CW | SPSA | HSJA | DeepFool |
|---|---|---|---|---|---|---|
| Gradient-based | ✓ | ✓ | ✓ | × | × | × |
| Iterative method | × | ✓ | ✓ | ✓ | ✓ | ✓ |
| Norm | $l_1, l_2, l_\infty$ | $l_2, l_\infty$ | $l_2$ | $l_1, l_2, l_\infty$ | $l_2, l_\infty$ | $l_2$ |

Table 2: Classification of the several attacks

## 4.2 Impact of the adversarial example for the training

We compare in Figure 8 the accuracy of the three different training methods using clean samples and FGSM and PGD adversarial examples from the test datasets. We can see for the basic training in Figure 8a that the accuracy is poor for the adversarial examples. However, the fact that the accuracy is decreasing and that it is not constantly bad can be explained by on the one hand, over the epochs the models converge towards a model specialized on the input data, making its gradient more informative and thus making it easier to attack. On the other hand, as from the first epochs, the model can misclassify a sample and an attack can accidentally make it predict the correct label. For adversarial training with FGSM and PGD in Figure Figure 8b and Figure 8c that the convergence is similar and the accuracy is slightly lower for the adversarial examples than the clean samples.



(a) Basic training

(b) FGSM adversarial training

(c) PGD adversarial training

Figure 8: Accuracy using clean samples and adversarial examples crafted with FGSM and PGD from the test samples for the models trained with basic training and FGSM and PGD adversarial training using the fast training code

In Figure 9 and Figure 10 we compare, for the fast and the basic training, the convergence of the accuracy on the train samples — as this is the only available accuracy for the basic training, on the FashionMNIST and CIFAR10 datasets. For the clean samples, the difference between the fast and basic training is the use of a cyclical learning rate. For FashionMNIST, the convergence on the clean samples is slightly better and for the FGSM adversarial examples, the number of epochs does not allow a trend to be determined. However, for the CIFAR10 the basic training is achieve a better convergence in both cases. This could be explained by the choice of the bounds chosen for the cyclical learning rate It seems that the default value of the learning rate of the optimize performs better than an advanced technique which is supposed to avoid performing hyperparameter tuning but does not work out of the box. So fine-tuning seems necessary to properly exploit the fast training. However, for the adversarial training, the accuracy stagnates which can be explained by the fact that the problem is too hard for this simple model.

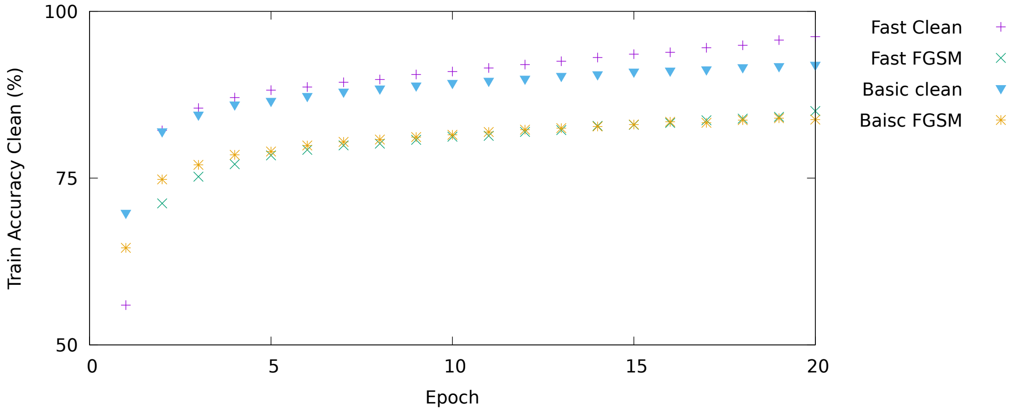


Figure 9: Comparison of the accuracy on the training samples use on the corresponding epoch between the basic and the fast train with the clean samples and FGSM adversarial examples on FashionMNIST

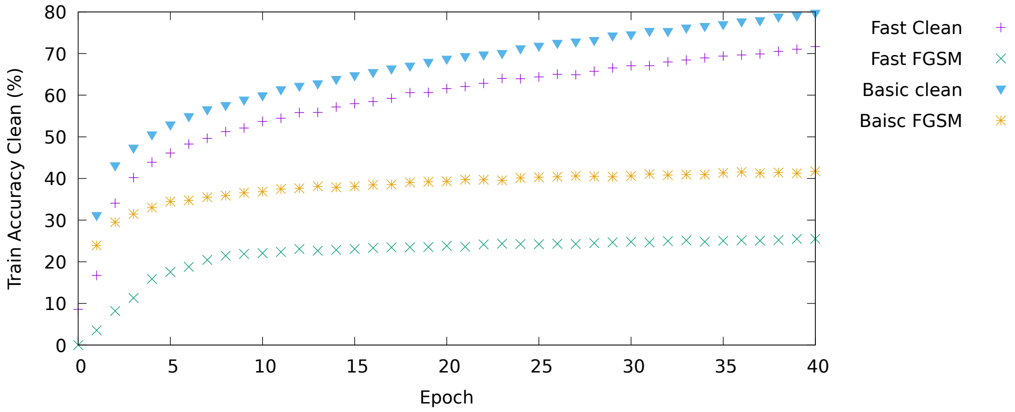


Figure 10: Comparison of the accuracy on the training samples use on the corresponding epoch between the basic and the fast train with the clean samples and FGSM adversarial examples on CIFAR10

## 4.3 Evaluating the model's accuracy

We compare in Table 3 the accuracy of the same previous model against a range of attacks on FashionMNIST. It enlightens the fact that adversarial training leads to a small loss of accuracy of the clean samples. However, for the gradient-based attacks — FGSM, PGD, CW, the accuracy is clearly better for an adversarially trained model by a factor of up to 6. Nevertheless, for the non-gradient-based attacks, the performance is still catastrophic for HSJA and DeepFool. The accuracy is really good against SPSA which can be explained by the fact that the parameters of the attack do not allow the attack to be strong enough. For the test on CIFAR10 in Table 4 the global trend is maintained, but as seen previously in Figure 10 the overall accuracy of the adversarial training is poor.

| Attacks | Acurracy | | |
|---|---|---|---|
| names | Clean | FGSM | PGD |
| Clean | 91.54 | 88.90 | 88.79 |
| FGSM | 28.08 | 84.92 | 84.89 |
| PGD | 20.73 | 82.38 | 83.43 |
| CW | 12.37 | 73.43 | 73.92 |
| SPSA | 85.81 | 88.54 | 88.30 |
| HSJA | 3.860 | | 5.029 |
| Deepfool | 6.300 | 7.970 | 7.990 |

Table 3: Comparison of the model's accuracy against some attacks between the training with clean samples and with PGD adversarial examples using the fast training code on FashionMNIST

| Attacks | Acurracy | |
|---|---|---|
| names | Clean | PGD |
| Clean | 72.59 | 37.81 |
| FGSM | 12.44 | 32.47 |
| PGD | 10.34 | 31.81 |
| CW | 16.13 | 36.09 |
| SPSA | 66.61 | 37.25 |
| HSJA | 10.38 | 16.53 |
| Deepfool | 14.52 | 19.43 |

Table 4: Comparison of the model's accuracy against some attacks between the training with clean samples and with PGD adversarial examples using the fast training code on CIFAR10

We now assess the impact of the $\varepsilon$ value of the attack on a model trained with attacks using a fixed value of $\varepsilon = 10$ on FashionMNIST. We can see for the training on clean samples in Figure 11 that the accuracy quickly decreases and then stagnates. For the adversarial training, the PGD training is slightly more accurate but for both the accuracy quickly decreases for $\varepsilon$ higher than the one used for the training (10).

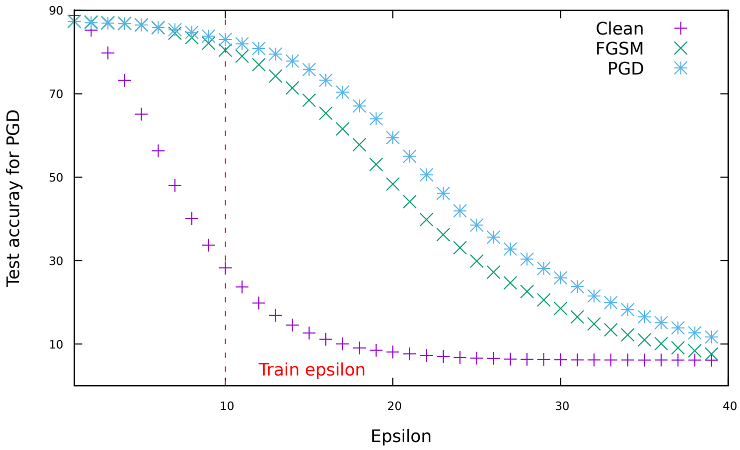


Figure 11: Evolution of the accuracy for the PGD attack for several $\varepsilon$ values on the test samples on the model trained with clean samples and with FGSM and PGD adversarial examples with $\varepsilon = 10$ with the basic training code on FashionMNIST

The Figure 12 illustrates an attack using PGD on a FashionMNIST, the trouser is misclassified as a T-shirt. We can see that the perturbation modifies around 65% pixels of the image with an intensity of ±0.06 for pixels values between 0 to 1.

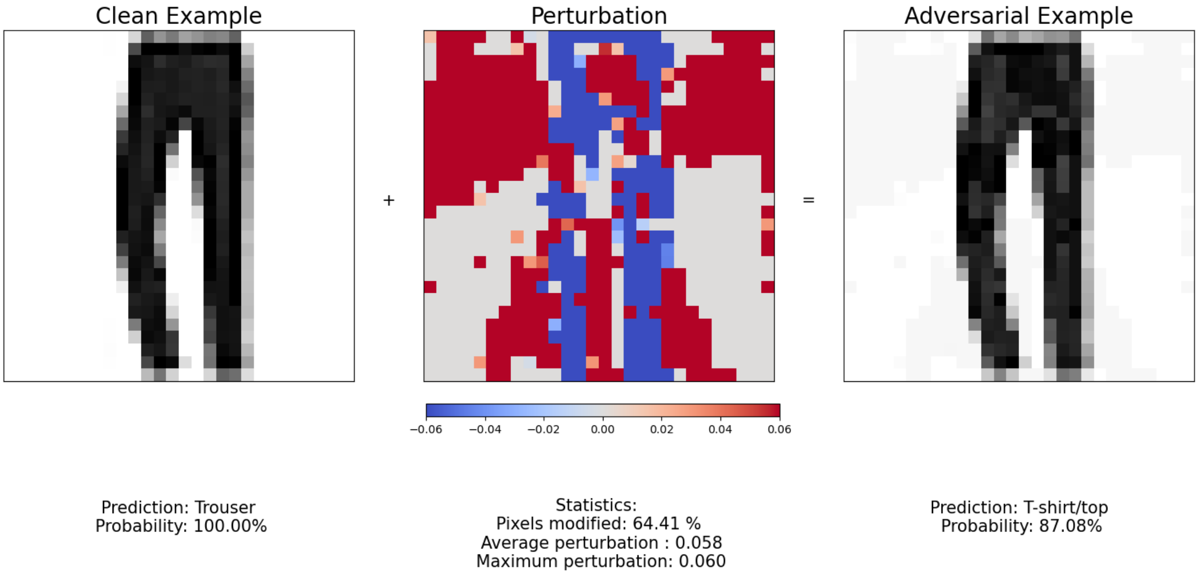


Figure 12: Example of a PGD adversarial attack on a FashionMNIST image

## 4.4 Confidence Learning

This experiment aimed to see if confidence learning, which has proven to be useful at predicting out-of-distribution (OOD) examples as such, could also be used to make a neural network robust against adversarial examples, which could be considered as OOD examples.

The first step consists of modifying the network in order to also output a confidence bit, this results in the architecture presented in Figure 13 :

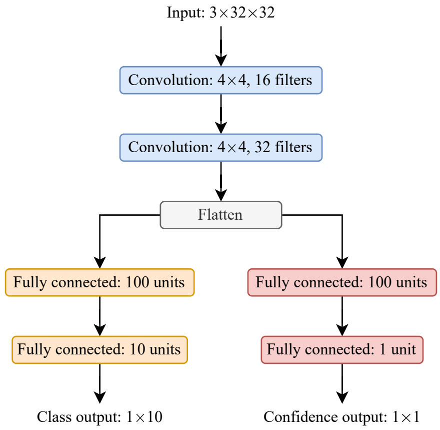
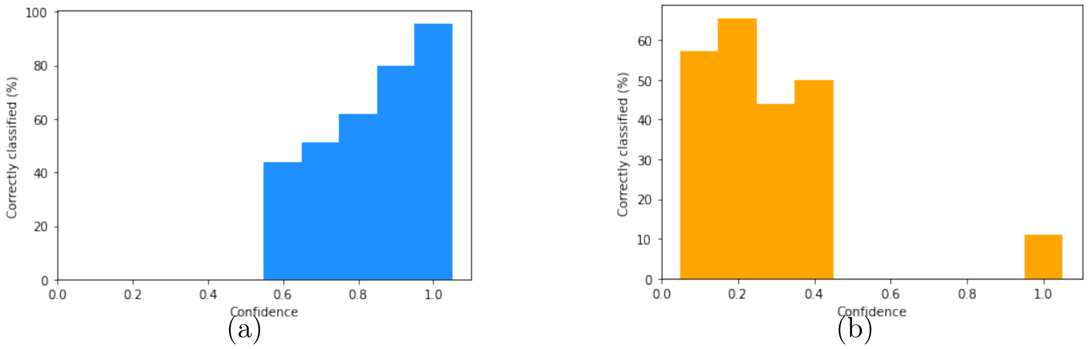

Input: 3×32×32
Convolution: 4×4, 16 filters
Convolution: 4×4, 32 filters
Flatten
Fully connected: 100 units
Fully connected: 100 units
Fully connected: 10 units
Fully connected: 1 unit
Class output: 1×10
Confidence output: 1×1

Figure 13: The "small" CNN representation with an added confidence prediction branch.

The next step is to train this network on the dataset, here FashionMNIST, using the loss formulated in Eq. 13 to train the confidence prediction alongside class prediction. The trained model is well calibrated on clean examples, as its confidence follows the percentage of correctly classified examples with the given confidence (Figure 14a), it also has a correct accuracy (Tab. 5). Nevertheless, on adversarial examples the model is no more well calibrated, correctly classifying examples with high uncertainty or misclassifying them with high confidence, the overall accuracy of the model also drops on adversarial examples (Tab. 5).

| | Accuracy | Average Confidence |
|---|---|---|
| Clean | 83.3 | 87.2 |
| FGSM | 16.1 | 67.5 |
| PGD | 18.8 | 84.8 |

Table 5: Accuracy and confidence (in %) of the network on Clean, FGSM, and PGD examples.



(a) (b)

Figure 14: Percentage of correctly classified clean images with respect to the confidence output for clean (14a) and adversarial (14b) examples.

## 4.5 Verification

### 4.5.1 Neural networks

To verify neural networks we used $\alpha - \beta - CROWN$ which implements complete and incomplete verification in python using bound propagation. For some features it requires licence optimizers (Gurobi and CPlex). This tool is very complete having multiple verification type and adding custom models is straightforward.

The experiment consisted in evaluating the performance of $\alpha - \beta - CROWN$ on three common classification data sets (MNIST, FashionMNIST, CIFAR-10) with three models based on `cnn_small` (a small convolution neural networks of 160 thousands parameters). `Clean` was trained in a standard way, `FGSM` was trained with adversarial `FGSM` attacks, `PGD` was trained with adversarial `PGD` attacks. Then the verifier was run on 100 inputs, giving the following results:

| Data sets | Verified accuracy | | | Execution time in second | | |
|---|---|---|---|---|---|---|
| | Clean | FGSM | PGD | Clean | FGSM | PGD |
| MNIST | 61 | 97 | 97 | 3480 | 54 | 325 |
| FashionMNIST | 9 | 60 | 73 | 1118 | 1536 | 325 |
| CIFAR-10 | 0 | 0 | 0 | 39 | 287 | 813 |

Table 6: Percentage of safe input (Verified accuracy) depending on the model and the data set

On small data sets (MNIST, FashionMNIST), robusts training worked really well, giving a higher number of safe inputs with better adversarial training. For CIFAR-10, our base model was not big enough to give good results even in classic classification.

### 4.5.2 Weight sparsity and ReLU stability

We studied methods presented by Xiao et al. [32] and tested them in contexts not presented in the original paper. (For more details on the code see Appendix D)). In each case, three models were trained, the results presented below are the average of the observed results. The small size of each sample is due to the fact that we have limited resources. It is not a problem in general given the proximity of the measured results except for the case of the ReLu pruning in the first layer where we observed variations from simple to double.

In Table 7, it is clear that the different techniques used to reduce the complexity of systems solved to verify models do not affect accuracy. The accuracy for naturally trained model on MNIST with $\epsilon = 10/255$ is unexpectedly high. Nevertheless, the fact that $\ell_1$-regularization and RS Loss are methods meant to be used in addition to adversarial training and that their weight in the loss has been optimized for $\epsilon$ near 0.1 can explain this good accuracy.

| Data set | $\epsilon$ | Adv. training | Training with $\ell_1$-reg + RS Loss | | Post process | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | Weight prune | | ReLU prune | |
| | | | Clean | PGD | Clean | PGD | Clean | PGD |
| MNIST | 10/255 | yes | 97.64 | 94.17 | 97.65 | 94.18 | 97.55 | – |
| | | no | 97.00 | 82.37 | 97.04 | 82.74 | 97.02 | – |
| | 0.1 | yes | 97.29 | 89.20 | 97.30 | 89.25 | 97.19 | – |
| | | no | 97.17 | 24.59 | 97.22 | 24.52 | 97.07 | – |
| FashionMNIST | 0.1 | yes | 76.65 | 60.71 | 76.73 | 61.03 | – | – |
| | | no | 85.88 | 14.60 | 85.96 | 14.55 | – | – |

Table 7: Models' accuracy (in %) at each processing step in different settings

We have not been able to compute clean accuracy for FashionMNIST nor PGD accuracies post ReLU pruning due to issue with the verifier used (see Appendix D).Additionnaly, generated systems could not be solved nor their size be extracted. Thus, the next two tables will present the impact of weight and ReLU pruning on their respective total.

Table 8 shows the number of weights left after weight pruning. Despite the fact that models' accuracy presented in Table 7 do not vary significantly after weight pruning, this table shows that in general 25% of weights were pruned. More precisely, less than half of weight of the first layer, a fourth of the second and 90% of the third were left untouched. We can also see that in general, less weights are pruned in adversarially trained models. Intuitively, this comes naturally since the adversarial training force the model to learn a more detailed approximation of the classification.

| Data set | $\epsilon$ | Adv. training | Layer1 | Layer2 | Layer3 |
|---|---|---|---|---|---|
| MNIST | 10/255 | yes | 150.67 | 2732.33 | 123745.67 |
| | | no | 107.33 | 2026.00 | 124560.67 |
| | 0.1 | yes | 196.00 | 3640.33 | 129087.00 |
| | | no | 102.67 | 1933.33 | 123970.33 |
| FashionMNIST | 0.1 | yes | 167.00 | 2774.00 | 126489.67 |
| | | no | 116.00 | 2775.33 | 131670.00 |
| Total | | | 400 | 12800 | 156800 |

Table 8: Number of remaining non zero weights in each layer of the NN after weight pruning

Table 9 shows the number of ReLU functions which have not been replaced either by the null function or the identity. We can see that for MNIST, less than 20% of ReLUs are still present meaning that RS Loss is very effective at encouraging ReLU stability. For FashionMNIST, this number drops to 30%, this is probably due to the fact that the weight of RS Loss in the loss fnction has been optimized for MNIST. Finally, the same remark about the difference between naturally and adversially trained models on the previous table holds here again.

| Data set | $\epsilon$ | Adv. training | Layer1 | Layer2 | Layer3 |
|---|---|---|---|---|---|
| MNIST | 10/255 | yes | 402.00 | 637.33 | 48.33 |
| | | no | 119.00 | 501.00 | 58.50 |
| | 0.1 | yes | 739.00 | 511.67 | 40.67 |
| | | no | 108.33 | 499.00 | 52.33 |
| FashionMNIST | 0.1 | yes | 1284.33 | 538.33 | 36.00 |
| | | no | 290.00 | 608.00 | 50.33 |
| Total | | | 3136 | 1568 | 100 |

Table 9: Number of ReLU left in each layer of the NN after ReLU pruning

### 4.5.3 Decision trees

The tool used here is $treeVerification$ [11]. Tests were made the same way as for neural networks, but since the models are trees, the adversarial training was quite different and made with a software from the same authors [8]. The data sets were the same except for one, and the accuracy that is possible to lose with attacks on the 100 inputs provided was compared:

[8] https://github.com/chenhongge/RobustTrees

| Data | Maximum possible loss of precision | | Execution time in second | |
|---|---|---|---|---|
| sets | Clean | Robust | Clean | Robust |
| MNIST | 100 | 0 | 40 | 86 |
| FashionMNIST | 100 | 30 | 46 | 73 |

Table 10: Maximum possible loss of precision depending on data set and model

The results are similar to those for neural networks, meaning that the adversarial training was highly efficient. The running time for unrobust model is taking

### 4.5.4 Other tools

Please find in Appendix the list of tools that we tried with their operating status.

## 4.6 Explainability

### 4.6.1 SHAP and LIME

Those tools are based on Shapeley values, a concept that comes from Game's Theory. Those tools are described as feature attribution methods, as they assign an importance grade to each feature of an input.

LIME is much more documented than SHAP, and comes with various notebook tutorials, making it the most viable option between both.

However, even if they can be really efficient in certain examples, those tools cannot be trusted for two reasons :

- Both tools compute approximate shapeley values, as computing the exact value is time-consuming.
- Shapeley values aren't formally viable to explain every feature as their explanations are local, and certain well-crafted inputs can show the limits of this method.

### 4.6.2 Anchor

Anchor is a tool described as a feature selection method. "An anchor explanation is a rule that sufficiently 'anchors' the prediction locally – such that changes to the rest of the feature values of the instance do not matter" Thus, we can have multiple anchor explanations for one input.

This method seems promising but could be subject to the same problem as SHAP and LIME which have unstable formal foundations.

## 5 Conclusion

# Appendices

## Appendix A Reproduce the experiments

**For the basic training:** Batch size = 512, Epochs = 20 for Fashion MNIST 40 for CIFAR10, Epislon = 0.0392 (=10/255), PGD: eps_iter = 0.01 , iter = 40

**For the fast training:** The parameters used are dumped in the first line of the corresponding log file of a training for instance:

Namespace(batch_size=256, data_dir='cifar-data', dataset='CIFAR10', model='cnn_small', epochs=40, lr_schedule='cyclic', lr_max=0.003, attack='fgsm', eps=10.0, attack_iters=10, pgd_train_n_iters=10, pgd_alpha_train=2.0, fgsm_alpha=1.25, minibatch_replay=1, weight_decay=0.0005, attack_init='zero', fname='plain_cifar10', seed=0, gpu=0, debug=False, half_prec=False, grad_align_cos_lambda=0.0, eval_early_stopped_model=False, eval_iter_freq=50, n_eval_every_k_iter=256, n_layers=1 (unused), n_filters_cnn=4 (unused), batch_size_eval=256, n_final_eval=1000)

**For the attacks:** the parameters of the attacks used for to assess the accuracy of the models are available in the configuration file `config_file/evaluation.cfg`

**For the verifications:**

- Neural network: for `CROWN` all the parameters are described in configuration files in $\alpha - \beta$`-CROWN/complete_verifier/exp_configs/tutorial_examples/"config".yaml`
- Decicion tree: all the parameters are described in `treeVerification/config_"dataset"_"training".json`

## Appendix B Run the project

The adversarial training and attacks environment is based on Python 3.10, it is created from conda. The full steps to create this environment is described on the project's github. Some dependencies are described below. The full dependencies are available in `env_files/conda_info.txt` and `env_files/full_conda_env.txt`

**Python**: 3.10.9
**conda**: 23.1.0

| Package | Version |
|---|---|
| pytorch | 1.13.1 |
| pytorch-cuda | 11.7 |
| torchvision | 0.14.1 |
| numpy | 1.23.5 |

Table 11: Package versions

The neural network verification makes use of Python 3.7 and also relies conda. The list of dependencies are available in `alpha-beta-CROWN/complete_verifier/environment.yml`. The feature requiring licences (Gurobi and CPlex) were not used in our project. As for the tree verifier, `libuv` and `libboost` are required to compile the tool but it is already compiled in our repository.

# Appendix C Tool list

| Name | State | Comments |
|---|---|---|
| $\alpha - \beta - CROWN$ [a] | usable | CPLEX and Gurobi for full features |
| MN-BaB [b] | unusable | password for gitlab submodule required |
| VeriNet [c] | unusable | Xpress licence required |
| nnenum [d] | usable | no issues |
| CGDTest [e] | unusable | repository not reachable (404) |
| PeregriNN [f] | unusable | a file was missing |
| Marabou [g] | unusable | compilation issues |
| Debona [h] | usable | no explanation so arguments hard to determine |
| FastBATLLNN [i] | unusable | no readme and install script does not work |
| treeVerification [j] | usable | works straightforward |

Table 12: List of tools for verification

[a] https://github.com/Verified-Intelligence/alpha-beta-CROWN
[b] https://github.com/eth-sri/mn-bab
[c] https://github.com/vas-group-imperial/VeriNet
[d] https://github.com/stanleybak/nnenum
[e] https://github.com/vin-nag/CGD.git
[f] https://github.com/haithamkhedr/PeregriNN/tree/vnn2022
[g] https://github.com/NeuralNetworkVerification/Marabou
[h] https://github.com/ChristopherBrix/Debona
[i] https://github.com/jferlez/FastBATLLNN-VNNCOMP
[j] https://github.com/chenhongge/treeVerification

# Appendix D Weight sparsity and ReLU stability

We directly used the code of the authors available on `GitHub`[9]. However, the code uses deprecated functions of TensorFlow and needed to be updated to use the proposed solver, `MIPVerify`[10]. This updated version along with the details of the parameters used in the experiments can be found on our `GitHub` repository under the folder `relu_stability`. Additionally, despite our best effort, due to a lack of documentation, we were not able to verify trained models because the solver crashes when reaching time limit and we are not sure if this comes from `MIPVerify`, `JuMP`[11] or `Gurobi`[12].

We also tried without success to convert trained models checkpoints in `.onnx`, a format accepted by $\alpha - \beta$-`CROWN` with `tf2onnx`[13].

Overall, it is probably better to reimplement from the beginning the ideas of Xiao et al. [32].

[9] https://github.com/MadryLab/relu_stable

[10] https://github.com/vtjeng/MIPVerify.jl

[11] https://jump.dev/JuMP.jl/stable/

[12] https://www.gurobi.com/

[13] https://github.com/onnx/tensorflow-onnx/

# References

[1] Dan Cireşan, Ueli Meier, Jonathan Masci, and Jürgen Schmidhuber. Multi-column deep neural network for traffic sign classification. Neural Networks, 32:333–338, 2012. ISSN 0893-6080. doi: https://doi.org/10.1016/j.neunet.2012.02.023. URL https://www.sciencedirect.com/science/article/pii/S0893608012000524. Selected Papers from IJCNN 2011.

[2] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks, 2013. URL https://arxiv.org/abs/1312.6199.

[3] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples, March 2015. URL http://arxiv.org/abs/1412.6572.

[4] Han Xu, Yao Ma, Haochen Liu, Debayan Deb, Hui Liu, Jiliang Tang, and Anil K. Jain. Adversarial Attacks and Defenses in Images, Graphs and Text: A Review, October 2019.

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

[6] Zeshan Kurd and Tim Kelly. Establishing Safety Criteria for Artificial Neural Networks. In Knowledge-Based Intelligent Information and Engineering Systems, volume 2773, pages 163–169, September 2003. ISBN 978-3-540-40803-1. doi: 10.1007/978-3-540-45224-9_24.

[7] Caterina Urban and Antoine Miné. A Review of Formal Methods applied to Machine Learning, April 2021.

[8] Rudy Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and M. Pawan Kumar. A Unified View of Piecewise Linear Neural Network Verification, May 2018.

[9] Vincent Tjeng, Kai Xiao, and Russ Tedrake. Evaluating Robustness of Neural Networks with Mixed Integer Programming, February 2019.

[10] Caterina Urban, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Perfectly Parallel Fairness Certification of Neural Networks, April 2020.

[11] Hongge Chen, Huan Zhang, and Cho-Jui Hsieh. Robustness Verification of Tree-based Models. page 12, 2019.

[12] Mark Niklas Müller, Christopher Brix, Stanley Bak, Changliu Liu, and Taylor T. Johnson. The Third International Verification of Neural Networks Competition (VNN-COMP 2022): Summary and Results, December 2022.

[13] Nicholas Carlini and David Wagner. Towards Evaluating the Robustness of Neural Networks, March 2017. URL http://arxiv.org/abs/1608.04644.

[14] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards Deep Learning Models Resistant to Adversarial Attacks, September 2019. URL http://arxiv.org/abs/1706.06083.

[15] Ali Shahin Shamsabadi, Ricardo Sanchez-Matilla, and Andrea Cavallaro. ColorFool: Semantic Adversarial Colorization, April 2020.

[16] Jianbo Chen, Michael I. Jordan, and Martin J. Wainwright. Hopskipjumpattack: A query-efficient decision-based attack. In 2020 IEEE Symposium on Security and Privacy (SP), pages 1277–1294, 2020. doi: 10.1109/SP40000.2020.00045.

[17] Jonathan Uesato, Brendan O'Donoghue, Aaron van den Oord, and Pushmeet Kohli. Adversarial risk and the dangers of evaluating against weak attacks, 2018. URL https://arxiv.org/abs/1802.05666.

[18] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. Commun. ACM, 63(11):139–144, oct 2020. ISSN 0001-0782. doi: 10.1145/3422622. URL https://doi.org/10.1145/3422622.

[19] Chaowei Xiao, Bo Li, Jun-Yan Zhu, Warren He, Mingyan Liu, and Dawn Song. Generating adversarial examples with adversarial networks, 2018. URL https://arxiv.org/abs/1801.02610.

[20] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in Machine Learning: From Phenomena to Black-Box Attacks using Adversarial Samples, May 2016.

[21] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. Parametric Noise Injection: Trainable Randomness to Improve Deep Neural Network Robustness against Adversarial Attack, November 2018.

[22] Cihang Xie, Yuxin Wu, Laurens van der Maaten, Alan Yuille, and Kaiming He. Feature Denoising for Improving Adversarial Robustness, March 2019.

[23] Eric Wong, Leslie Rice, and J. Zico Kolter. Fast is better than free: Revisiting adversarial training, 2020. URL https://arxiv.org/abs/2001.03994.

[24] Maksym Andriushchenko and Nicolas Flammarion. Understanding and improving fast adversarial training, 2020. URL https://arxiv.org/abs/2007.02617.

[25] Leslie N. Smith. Cyclical learning rates for training neural networks. In 2017 IEEE Winter Conference on Applications of Computer Vision (WACV), pages 464–472, 2017. doi: 10.1109/WACV.2017.58.

[26] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. On Calibration of Modern Neural Networks, August 2017.

[27] Pavel Gurevich and Hannes Stuke. Pairing an arbitrary regressor with an artificial neural network estimating aleatoric uncertainty, September 2018.

[28] Terrance DeVries and Graham W. Taylor. Learning Confidence for Out-of-Distribution Detection in Neural Networks, February 2018.

[29] M. A. Ganaie, Minghui Hu, A. K. Malik, M. Tanveer, and P. N. Suganthan. Ensemble deep learning: A review. Engineering Applications of Artificial Intelligence, 115: 105151, October 2022. ISSN 0952-1976. doi: 10.1016/j.engappai.2022.105151.

[30] Maximilian Baader, Matthew Mirman, and Martin Vechev. Universal Approximation with Certified Networks, January 2020.

[31] Zi Wang, Aws Albarghouthi, Gautam Prakriya, and Somesh Jha. Interval Universal Approximation for Neural Networks. Proceedings of the ACM on Programming Languages, 6(POPL):1–29, January 2022. ISSN 2475-1421. doi: 10.1145/3498675.

[32] Kai Y. Xiao, Vincent Tjeng, Nur Muhammad Shafiullah, and Aleksander Madry. Training for Faster Adversarial Robustness Verification via Inducing ReLU Stability, April 2019.

[33] Joao Marques-Silva and Alexey Ignatiev. Delivering Trustworthy AI through formal XAI. In Proc. of AAAI, pages 3806–3814, 2022. URL https://scholar.google.com/citations?view_op=view_citation&hl=en&user=1b9hppwAAAAJ&sortby=pubdate&citation_for_view=1b9hppwAAAAJ:KS-xo-ZNxMsC.

[34] Joao Marques-Silva. Logic-Based Explainability in Machine Learning, October 2022.

[35] Thomas Fel, Melanie Ducoffe, David Vigouroux, Remi Cadene, Mikael Capelle, Claire Nicodeme, and Thomas Serre. Don't Lie to Me! Robust and Efficient Explainability with Verified Perturbation Analysis. arXiv, February 2022.

[36] Joel Vaughan, Agus Sudjianto, Erind Brahimi, Jie Chen, and Vijayan N. Nair. Explainable Neural Networks based on Additive Index Models, June 2018.

[37] Zebin Yang, Aijun Zhang, and Agus Sudjianto. Enhancing Explainability of Neural Networks through Architecture Constraints, September 2019.

[38] Vugar Ismailov. Notes on ridge functions and neural networks, August 2020.

[39] Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian Goodfellow, Aleksander Madry, and Alexey Kurakin. On Evaluating Adversarial Robustness, February 2019.

[40] Yaxin Li, Wei Jin, Han Xu, and Jiliang Tang. DeepRobust: A PyTorch Library for Adversarial Attacks and Defenses, May 2020.

[41] Nicolas Papernot, Fartash Faghri, Nicholas Carlini, Ian Goodfellow, Reuben Feinman, Alexey Kurakin, Cihang Xie, Yash Sharma, Tom Brown, Aurko Roy, Alexander Matyasko, Vahid Behzadan, Karen Hambardzumyan, Zhishuai Zhang, Yi-Lin Juang, Zhi Li, Ryan Sheatsley, Abhibhav Garg, Jonathan Uesato, Willi Gierke, Yinpeng Dong, David Berthelot, Paul Hendricks, Jonas Rauber, Rujun Long, and Patrick McDaniel. Technical Report on the CleverHans v2.1.0 Adversarial Examples Library, June 2018. URL `http://arxiv.org/abs/1610.00768`.

[42] Jonas Rauber, Wieland Brendel, and Matthias Bethge. Foolbox: A Python toolbox to benchmark the robustness of machine learning models, March 2018.

[43] Hongge Chen, Huan Zhang, Duane Boning, and Cho-Jui Hsieh. Robust decision trees against adversarial examples. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, Proceedings of the 36th International Conference on Machine Learning, volume 97 of Proceedings of Machine Learning Research, pages 1122–1131. PMLR, 09–15 Jun 2019.