UCLouvain

epl

École polytechnique de Louvain

Verifying Binarized Neural Networks using Marabou

Author: **Maciej Piotrowski**
Supervisor: **Siegfried Nijssen**
Readers: **Pierre Schaus, Cyril De Bodt**
Academic year 2022–2023
Master [120] in Computer Science

Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my heartfelt gratitude to my supervisor, Professor Siegfried Nijssen, for his invaluable guidance, support, and encouragement throughout the course of my research. His patience, wisdom, and expertise have been an invaluable resource and have helped shape this work into what it is today.

I would also like to thank my parents for their unwavering support and encouragement throughout my academic journey. Their love and belief in me have been a constant source of motivation and have helped me persevere through the challenges of my studies.

I am deeply grateful to my friends for their support and encouragement during this journey. Their encouragement and good cheer have helped me through the many ups and downs of university and have made this experience all the more enjoyable.

I would also like to thank Guy Amir for providing me with access to the work that he had done before and for the clarification he made concerning the framework.

Finally, I would like to extend my thanks to Benoit Ronval for generously sharing his insights and observations on the subject of binarized neural networks. His ideas and remarks have been invaluable and have greatly enhanced the quality of this work.

Overall, this work would not have been possible without the support and guidance of all of these wonderful people. I am deeply grateful for their help and encouragement and am truly thankful to have had the opportunity to work with them.

Chapter 1

Introduction

For many years, artificial intelligence has been part of the computer science domain. Though initially viewed with great optimism, interest in the field dwindled in the 1970s and 1980s. This period, known as the "AI winter," occurred due to a variety of causes, including the inability of various AI initiatives to deliver on their promises and the technological limitations at the time. However, we have seen a significant increase in interest in artificial intelligence in recent years, and this is not without reason. In the first decades of the 21st century, with more powerful machines at our disposal, machine learning emerged. It is a highly mathematical-statistical technique that has proven to be quite effective in resolving several difficult issues that had previously remained unresolved.

One specific algorithmic class in the broad family of machine learning methods is deep learning. It is a topic that has grown at an exponential rate in the last decade, particularly with regard to deep neural networks. Deep neural networks achieved astonishing improvements in a wide range of tasks like image recognition, speech recognition, and even playing games. It is highly configurable with a big variety in terms of its components: neurons, synapses, weights, biases, and activation functions. However, those networks are posing some challenges, among which we can find the need for huge computational power and the difficulty of exactly verifying those kinds of networks. We will try to investigate how we can solve issues related to the exact verification in this thesis.

The first issue is a well-known one. Deep neural networks are able to achieve great results, but to do so, they are trained on graphics processing units that are very fast and power-hungry. It has its limitations, and the main research that is done to tackle that problem is to speed up deep neural networks at run time and build specialized computer hardware. It's not perfect because that kind of solution is inconceivable from a green computing point of view, which, as we know,

in today's world is a topic that we shouldn't ignore. It is inconceivable because the production and usage of this specialized equipment both consume a lot of energy. Secondly, access to and use of this technology may be challenging for academics and organizations with low funding due to the high cost of the specialized hardware needed to run deep neural networks. Also, improving the hardware isn't always desirable. In the case of autonomous cars, it's useless to have big computers that achieve great results if they can't fit into a car. Or in the case of a smartphone's face recognition, it's useless to have really expensive graphics processing units if they will significantly raise the price of the smartphone. An alternative solution for that is to try to reduce the complexity of the calculations so that they become less demanding and can be run on smaller or cheaper devices, such as smartphones or nanocomputers.

The second issue is a bit more complicated. It's nice to have deep neural networks that can achieve great accuracy, but even with those, the question of verification still remains. At what point can we trust our models? This brings to the table the question of the network's robustness. Is it possible to add some perturbations, visible to the human eye or not, to an input on which our model performs well, so that the input is misclassified? This can apply to our previous examples as well. Do we want a car that sees a person as a shadow due to the weather conditions? Do we want our smartphones to be unable to recognize our faces due to the brightness of the room in which we are? Exact (complete and sound) verifiers have scaling concerns and cannot ensure correctness with deep neural networks because of floating-point mistakes.

We will try to investigate both issues in this thesis with the help of binarized neural networks [2, 3]. These are a special kind of deep neural network with run-time binary activations and weights. At training time, a common practice is to apply quantizers to the weights. This is required when using real-valued latent weights, a popular BNN optimization technique, to accumulate non-binary update steps and help compute the parameter gradients. The real-valued weights and related quantization procedures can be discarded after training is complete. As we are using only 1 bit to represent weights and output values, the computations done by this type of network are a lot less demanding, which partially solves our first issue since we can now run our models on smaller devices. But another issue arises: simpler computations inevitably mean a drop in precision. We'll try to figure out how much it affects model performance and compare different architectures. Are the models equivalent? Does the reduction in model size necessarily involve a significant decrease in accuracy? Can they handle problems of any complexity, and what are their limitations?

For the second issue, the verification of binarized neural networks is quite an emerging topic, and there has not been a lot of work done about it for the time being. However, we can already state that the binary format of weights and outputs allows us to build exact and significantly more efficient verifiers such as SAT and SMT-based solvers since we no longer face floating point precision errors. They are able to prove through formal verification that binarized neural networks can provide similar robustness. There is, for example, EEVBNN [4], a new SAT solver that handles the reified cardinality constraints that arise in binarized neural network encodings natively, accelerates binarized neural network verification, and consists of techniques for creating robust binarized neural networks that are amenable to solvers, such as balancing layer-wise sparsity and low cardinality constraints and adaptive gradient cancellation. Another available framework that we will focus on in this thesis is the Marabou [5] framework. Marabou is an open-source constraint satisfaction problem-answering tool that is based on SMT. It can answer queries regarding the attributes of a network by translating such queries into constraint satisfaction problems. Even though it's not originally dedicated for binarized neural networks, some recent work[6], integrated in the original distribution of Marabou, has extended the range of compatible networks, allowing the verification of binarized neural networks by, among other things, adding the sign constraint. It offers various optimizations as well as an approach for parallelizing verification queries. This framework will be used for the verification part of this thesis.

Regarding the various chapters, in Chapter 2, we will talk about the fundamental theoretical ideas needed to comprehend the different concepts discussed in this thesis, such as deep/binarized neural networks and their components, verification of models, and the different techniques that our framework can be used with. In Chapter 3, I will describe the different datasets used to train the models and their strengths and weaknesses. In Chapter 4, I will give a high-level overview of the different model architectures that will be used in further experiments and specify their training parameters.

In Chapter 5, I will perform experiments concerning the training of neural networks. In order to evaluate the impact of binarization of a model, I will compare the binarized neural networks to their traditional deep version from the point of view of the number of parameters, their sizes, their training time, and their accuracy. Then, depending on the dataset and their limitations, I will compare the under- and over-fitting behaviors of both models. Finally, I will show the importance of normalizing the data.

In Chapter 6, I will explain in more detail the methodology used by Marabou. We will go step by step through the whole process of verification, and I will explain all the parameters that are taken into account. I will explain the verification done by Marabou from a more formal point of view, and we will analyze the results that Marabou can produce. We will see that the traditional softmax function, often used in classification models, is not compatible with constraint verification tools such as Marabou, and we will try to overcome this issue and investigate the input queries that made that possible. We will evaluate the importance of the timeout and why we still need it. Lastly, I will introduce the concept of adversarial attack by region, which consists of fixing some pixels.

In the final chapter of this thesis, Chapter 7, I will investigate more in detail the impact of the bypass implemented for the softmax function. Are the results reliable, even with the inconvenience it introduces? We will perform experiments to assess the verification efficiency[5] and evaluate the performance difference of the SMT approach compared to the Gurobi solver, which is also integrated in the tool. I will discuss the potential trade-off between the complexity of a model, its accuracy, and its robustness, as this trade-off cannot be found in the current literature. And as the last topic of this thesis, we will focus on the adversarial examples that Marabou produces and see if we can observe some patterns between the different perturbations, if the verification is easy to perform, and what are the shortcomings related to the verification of a model? Does fixing some pixels help the solver or slow it down? Does one type of pixel have a greater impact than another?

Chapter 2

Concept definition

To fully grasp the unique characteristics of binary neural networks, we must first discuss some theoretical topics before moving on to the main points.

2.1 Deep Neural Networks

An artificial neural network is a type of machine learning algorithm modeled after the structure and function of biological neural networks that constitute the human brain. It is composed of a large number of interconnected processing nodes, called artificial neurons, that work together to solve complex problems. Unlike traditional algorithms, which require the programmer to specify a set of rules for solving a problem, a neural network is able to learn from examples and make predictions or decisions on its own. Nevertheless, a shallow neural network has only a single layer of neurons, which limits its ability to learn and model complex data.

In general, as we can see in the Figure 2.1, the simplest neural network, also called a perceptron, is composed of different parts:

- The input layer, made of neurons excited at a corresponding X-value.
- The weight vector W that represents weighted links between neurons including a bias.
- The activation function that is a weighted sum of inputs and their weights:

$$\sum_{i=0}^{n} w_i x_i$$

- The output that is computed after going through the activation function (here a binary step function corresponding to a thresholding function)

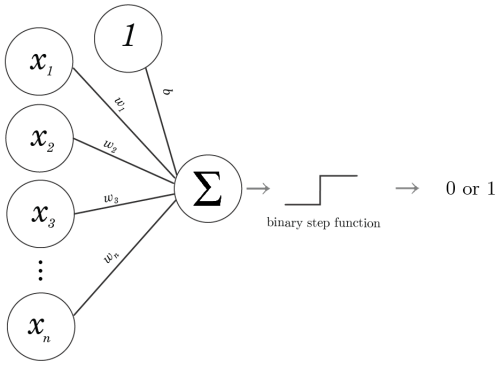

Figure 2.1: Single layer neural network

In contrast, a deep neural network is a type of neural network that has multiple layers of interconnected neurons, with the input layer at the bottom and the output layer at the top. This allows the network to learn and understand complex relationships between the input data and the desired output. Each layer's output is fed into the next layer, and so on, with each layer's output being calculated by a non-linear function of the total of its inputs. The inputs and outputs are real numbers. The strength of the connections between neurons, known as "weights", is adjusted during the learning process to optimize the performance of the network. It's possible to set a threshold for some neurons, and only when the total input exceeds that threshold will the neuron send out a signal. Those kinds of networks have been shown to be highly effective at a wide range of tasks, including image and speech recognition, natural language processing, and many others.

2.1.1 Input Normalization

Normalization is a preprocessing step that scales the values of the inputs to a common range, typically between 0 and 1. This can help improve the convergence of machine learning models and make the training process more efficient. There are several reasons why normalizing the inputs can be beneficial:

- Improved model convergence: normalizing the pixel values can help speed up the convergence of machine learning models, as it puts all the features on a common scale and reduces the range of the values. This can help the model converge faster and potentially improve its performance.
- Reduced model sensitivity to the scale of the input data: Normalizing the pixel values can also make the model less sensitive to the scale of the input

data. If the input values have different scales, this can be especially helpful because it can help the model learn more robust features that are independent of the scale of the input data.

- Improved model interpretability: Normalizing the pixel values can also make the model more interpretable, as it puts all the features on a common scale and makes it easier to compare the relative importance of different features.

The LARQ tutorial [7] uses a normalization of the pixel values to be between -1 and 1, but in my experiments I will use a range from 0 to 1 instead as it showed more satisfying results.

2.1.2 Batch Normalization Layer (BN)

Batch normalization [8] is a technique used in deep learning to improve the performance and stability of neural networks. It is typically used in the form of a layer within the network that normalizes the inputs so that they have a mean of zero and a standard deviation of one. This helps to reduce the internal covariate shift, which is the change in the distribution of the inputs that occurs during training. Batch normalization can improve the convergence of the network and cut down on the time it takes to train the network by reducing the internal covariate shift.

As stated in the paper [9], the batch normalizing transform applied to an activation x over a mini-batch of size m is defined as:

$$\hat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

where $\hat{x_i}$ are the different activations of the mini-batch, ϵ is a small constant added for numerical stability and μ_B and σ_B^2 are the mean and variance of the mini-batch, respectively:

$$\mu_B = \frac{1}{m} \sum_{i=1}^{m} x_i$$

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2$$

The transformed activations $\hat{x_i}$ are then scaled and shifted using learnable parameters γ and β:

$$y_i = \gamma \hat{x_i} + \beta$$

These parameters, when learned in conjunction with the parameters of the initial model, restore the representational capability of the network. The rest of the network uses the new activations y_i instead of the original activations x_i.

The distribution of the activations can become significantly skewed in binarized neural networks since the activations of the network can only be either -1 or 1. This might result in even more apparent issues with the network's ability to converge. Thus the need for batch normalization layers for binarized neural networks. There are several research papers published on the subject to demonstrate its effectiveness. [10]

2.1.3 Softmax activation function

I will now introduce the softmax activation function, as it is used by all of my binarized models to perform the classification.

The softmax activation function is a common choice for the output layer of a classification neural network. It turns a set of arbitrary real-valued activations into a probability distribution over a set of classes.

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Where x is a vector of real-valued activations and n is the number of classes. The output of the softmax function is a vector of values between 0 and 1 that sum to 1, representing the probability of each class. It will be really useful for us as we will mainly classify digit images, and that function will allow us to tell the probability for an image to belong to a specific class.

2.2 Binarized Neural Networks

Binarized neural networks (BNNs) are a type of neural network that uses binary values (in our case, -1 and 1) to represent the weights and activations of the network, rather than the real-valued weights and activations used in traditional neural networks. Thus are not to be confused with a binary activation network (BAN) where only the inputs are binarized or a binary weight network (BWN) which contains layers where only the kernels are quantized. The only non-binary layer is the first input layer.

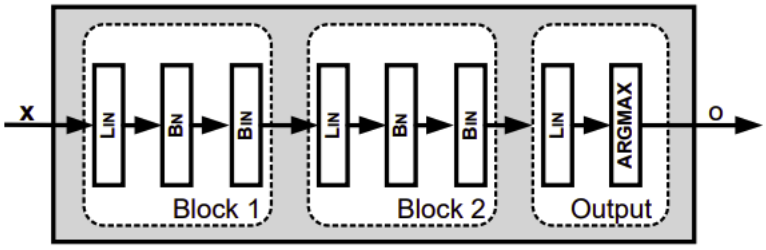

Figure 2.2: A schematic view of a binarized neural network [1]

As we can see in Figure 2.2, it is more convenient to talk about a binarized neural network's structure in terms of the composition of blocks of layers than it is to talk about individual layers. There are both linear and non-linear transformations in each block. Sequentially putting those blocks together results in a binarized neural network. A binarized neural network's internal block changes a binary input vector through a number of transformations before producing a binary output vector. Although a block's input and output are binary vectors, its internal layers can provide intermediate outputs with real values. Three major operations — a linear transformation (LIN), batch normalization (BN), and binarization (BIN) — are composing a typical internal block construction. The input vector is first transformed linearly. This affine linear transformation can be built as a convolutional layer or a fully connected layer. After the linear transformation, a scaling procedure using a batch normalization process is carried out. The sign function is then used to execute a binarization to produce a binary output vector. Bear in mind that the initial input and the final output of the whole network are the only values that are NOT binarized.

Binarized neural networks are typically much smaller and more efficient than traditional neural networks, as they require fewer bits to represent each weight and activation. This can make them more suitable for applications where memory and computational resources are limited, such as on mobile devices or a Raspberry Pi.

As explained in the introductory paper [2], to train a binarized neural network, the weights and activations of the network are typically passed through a binary thresholding function, which is often a simple deterministic sign function:

$$x^b = \text{sign}(x) = \begin{cases} +1 & \text{if } x \geq 0, \\ -1 & \text{if } x < 0, \end{cases}$$

x^b being the binarized variable. The resulting binary weights and activations can then be used to make predictions, just like in a traditional neural network. There

is also a stochastic alternartive for the binarization function :

$$x^b = \begin{cases} +1 & \text{with probability } p = \sigma(x), \\ -1 & \text{with probability } 1 - p, \end{cases}$$

where σ is a "hard sigmoid" function :

$$\sigma(x) = \text{clip}(\frac{x+1}{2}, 0, 1) = \max(0, \min(1, \frac{x+1}{2}))$$

Although the stochastic binarization is more interesting than the sign function, it is more difficult to implement since it needs the hardware to produce random bits when quantizing. I'll employ the first one in my experiments as a consequence.

2.2.1 Gradient

The real-valued gradients of the weights are accumulated in real-valued variables even though the binarized neural network training technique uses binary weights and activations to compute the parameter gradients. Real-valued weights are necessary for Stochasic Gradient Descent (SGD) to function at all. The stochastic gradient contributions accumulated in each weight help to average out the noise that results from SGD's exploration of the parameter space in brief, chaotic steps. As a result, it's crucial to maintain enough resolution for these accumulators.

Additionally, while computing the parameter gradients, introducing noise to the weights and activations results in a sort of regularization that can aid in greater generalization. When computing the gradients of the parameters during training, binarizing both the activations and the weights can be seen as an alternative to arbitrarily setting half of them to zero, as in dropout in classical neural networks.

Propagating gradients through discretization entails computing the gradients of the model's loss function with respect to the parameters and utilizing these gradients to modify the parameters during training. Since the weights and activations of a binarized neural network model may only be either +1 or -1, they cannot be differentiated, which is a big issue. That's why a straight-through estimator that takes into account the saturation effect, and uses deterministic rather than stochastic sampling is usually used.

2.3 Larq

Larq is an open-source deep learning Python library, developed by Plumerai [11], that provides tools and libraries for building and training binary neural networks. It was created as a way to use binary weights and activations rather than real-valued weights and activations to increase the effectiveness and durability of neural networks. One of the key benefits of Larq is that it allows developers to build and train binarized neural networks using familiar deep learning frameworks and tools, such as TensorFlow and Keras. Note also that it's possible to use Larq layers interchangeably with Keras layers because they are fully compliant with the Keras API.

Similar to how conventional neural networks are constructed, binarized neural networks can be created using a set of custom layers that LARQ offers. These layers comprise the QuantDense layer, which transforms the input data linearly using binary weights and activations, and the QuantConv2D layer, which transforms the data convolutionally using binary weights and activations. Tensorflow's dense and quantized 2D layers are equivalent in each case. A variety of activation functions are also supported by LARQ, including the sign function, which is the binarized neural network's default activation function, as well as ReLU and hard sigmoid. It is worth noting that the way LARQ structures a binarized neural network differs slightly from that shown in Figure 2.2. In LARQ, we have only 2 layers in an internal block : the linear transformation and the batch normalization layer. As a matter of fact, LARQ integrates binarization into its input and kernel quantizers, but at the end the result remains the same.

In addition to that, another part of the family of libraries for binarized neural network development is the LARQ Zoo library. It's a modeled zoo of ready-to-use, pre-trained models covering both state-of-the-art models on the ImageNet dataset as well as commonly used models in the literature. It may be extremely useful for researchers who either want to get started in the field and explore what is currently being done or researchers who want to develop a new training algorithm and want to build on top of current state-of-the-art models without spending hours trying to reproduce existing literature.

The last part is the LARQ Compute Engine, which is an optimized runtime for benchmarking and running neural networks on commonly used devices. It allows for evaluating binarized neural network performance as well as viewing the structure and behavior of the networks.

2.3.1 Quantizer

A quantizer in LARQ specifies how to convert a full-precision input to a quantized output and how to perform the backwards pass using the pseudo-gradient method. The latter is known as a "pseudo-gradient," since it typically does not represent a true gradient. In order to quantize activations, quantizers are often present across the network. This is due to the fact that the majority of layers sum over multiple binary values and output integers even when all inputs are binary..

Quantizers are frequently used on the weights while training. This is required when using real-valued latent weights, a popular binarized neural network optimization technique, to accumulate non-binary update steps. The real-valued weights and related quantization procedures can be discarded after training is complete. It is also possible to use some custom quantizers.

2.3.2 Quantized layers

Every quantized layer accepts an input quantizer and a kernel quantizer, which specify how to quantize the layer's weights and incoming activations, respectively. The layer is identical to a full precision layer if the input quantizer and kernel quantizer are both set to None. Both of them can be quantized independently in LARQ.

A quantized layer computes activations $\boldsymbol{y}$ as:

$$\boldsymbol{y}{=}\sigma(f(q_{kernel}(\boldsymbol{w}), q_{input}(\boldsymbol{x})) + b)$$

with $\boldsymbol{w}$ as full precision weights, $\boldsymbol{x}$ as arbitrary precision input, f as layer operation, σ as activation function and b as bias

2.4 Satisfiability Modulo Theories

The field of study known as SMT (Satisfiability Modulo Theories) [12, 13] focuses on the creation of tools and algorithms for determining whether a logical formula expressed in a specific logical theory is insatisfiable or whether there is a set of values that can be assigned to the formula's variables to make it true. Software tools known as SMT solvers are used to automatically solve satisfiability problems in a wide range of logical theories, including propositional logic, first-order logic, and many theories of arithmetic and data types. Automated theorem proving, software verification, and optimization are just a few of the various uses for SMT solvers.

In order to determine if a boolean formula is satisfiable, SMT solvers reduce the satisfiability problem to the boolean satisfiability (SAT) problem. This is accomplished by converting the logical formula into a boolean formula using techniques like expressing logical variables as boolean variables and encoding the logical operators as boolean functions. A SAT solver can then be used to solve the resulting boolean formula.

Because Boolean satisfiability is already NP-complete, the SMT problem is often NP-hard, and it is undecidable for many theories. Researchers investigate the computational complexity of decidable situations as well as which theories or subsets of theories give rise to a decidable SMT problem. Frequently, SMT solvers implement the resulting decision processes directly. One way to look at SMT is as a constraint satisfaction problem, and as such, it represents a certain formalized approach to constraint programming.

2.5 Marabou

Marabou [14] is an open-source constraint satisfaction problem answering tool that is based on SMT. It's actually a re-implementation of the Reluplex algorithm, which is basically the Simplex algorithm with native support for value constraints, augmented with lazy backtracking search. It is implemented in C++, although there is also a Python interface called Maraboupy, for which there is documentation [15] available that describes how to install it, provides examples of how to use it, explains how to contribute to Maraboupy, and provides API documentation. Marabou can answer queries about network properties by converting them into constraint satisfaction problems. It is able to work with networks that have a variety of activation functions and topologies, and it performs high-level reasoning on the network, which can reduce the amount of search space and enhance speed. In addition to this, it is capable of parallel execution (called Split&Conquer), which further improves scalability. Marabou is compatible with a wide variety of input formats, one of which is the protocol buffer file format, which is produced by the widely used TensorFlow framework for neural networks.

Marabou can support 2 different types of verification queries:

- Reachability queries: Is the output guaranteed to be in some, usually safe, range if the inputs are in the given range?
- Robustness queries: determine whether there exist adversarial points around a given input point that could alter the network's output.

In my thesis, I will focus on the robustness part, for which Marabou just takes as input a query and outputs either that the network is robust for the given inputs or provides a satisfying assignment for the constraints, which would essentially be a counter-example.

Marabou was used for a variety of verification tasks, such as network simplification and optimization, verification of video streaming protocols, deep neural network modification, adversarial robustness evaluation, verification of recurrent networks, and others. Although Marabou wasn't originally designed for binarized neural networks, a recent contribution [6] extends the framework to support the sign constraints and thus verify the robustness of binarized neural networks. This extension was approved and officially merged into the main Marabou repository. That's why we will use it for the experiments during my thesis. However, keep in mind that the traditional softmax function used in all my binarized neural networks models isn't compatible with constraint verification tools such as Marabou, and that in order to evaluate a network's output, we need to look at the operation just before the softmax layer.

2.6 Verification

The process of formally checking whether a neural network satisfies certain desired properties or behaviors, or returning a counter-example, is known as neural network verification [16, 17]. There are several possible applications of that :

- Check the robustness against adversarial examples [18, 19] : Formally prove that no matter how smart your adversarial attack is, it can never cause a misclassification in the network.
- Formally verify safety-critical systems that incorporate neural networks. For example: Guarantee that a drone control system won't cause any collisions.
- Extract network properties: make networks more understandable to humans.

In this thesis, we will focus on the first application. For the majority of machine learning problems, neural networks produce the most cutting-edge results. Inconveniently, neural networks are susceptible to attacks using adversarial examples. A possible approach to overcoming this challenge is formal verification, which formally certifies that networks are accurate. The majority of the neural network verification literature focuses on deep neural networks, for which exact verification is impossible. It's not an issue anymore for a binarized neural network, as all the weights are binarized, which tackles the problem of floating-point precision, and in

addition to that, the verification is significantly more efficient.

2.7 Gurobi

Gurobi [20] is a commercially available proprietary software program developed for the purpose of resolving optimization issues. It is capable of tackling very large and difficult optimization problems, as well as being designed to be highly efficient and precise. Several kinds of optimization issues, such as linear programming, quadratic programming, and mixed-integer programming, are all within Gurobi's scope of application as a tool for finding solutions. Gurobi has applications in a diverse set of industries, such as the financial sector, the energy sector, and the industrial sector. Portfolio optimization, optimization of supply chains, and scheduling are just a few of the applications that frequently make use of it. It is well-known for the high-performance optimization solvers it develops.

A wide range of programming languages, such as C, C++, C#, Java, Python, and R, are supported by Gurobi. It may be applied to the resolution of optimization issues inside a wide range of software platforms for optimization, including Marabou, among others. Support for multi-objective optimization, distributed parallel processing, and warm-starts are some of the more advanced capabilities and settings that can be found in Gurobi. Because it is a commercial software product, you will need to purchase a license in order to use it, or, if you're an academic, you can get a free, unlimited-use Gurobi Optimizer license on their website.

We will try to see how Marabou can benefit from Gurobi's advantages. We will only use the linear programming solver from Gurobi, as that's the one Marabou is using. Concerning linear programming for continuous models, Gurobi incorporates cutting-edge implementations of the most recent algorithms, such as primal and dual simplex algorithms, a parallel barrier approach with crossover, parallel optimization, and a shifting algorithm. These are only some of the examples. Gurobi incorporates a parallel barrier optimizer for linear problems; the barrier algorithms in Gurobi make full use of the advantages of the most recent computer architectures. Gurobi takes advantage of the widest variety of sophisticated presolve capabilities that are currently available in order to help simplify a given model, highlight any evident flaws or difficulties, and propose potentially beneficial ways for finding the best solution in the least amount of time.

2.8 Linear programming

In this section, I'll introduce the basics of linear programming, since both Marabou's native solver and the Gurobi solver are linear programming solvers. Gurobi can also be used as a MILP solver but we will not explore that path during this thesis.

A mathematical model whose requirements are represented by linear relationships can be optimized using a technique known as linear programming, sometimes known as linear optimization, to produce the best results (such as highest profit or lowest cost). Linear programming is a subset of mathematical programming (also known as mathematical optimization). Formally speaking, linear programming is a method for optimizing a linear objective function under the restrictions of linear equality and linear inequality.

Problems are considered to be linear programs if they are able to be represented in canonical form as :

$$\begin{aligned} \text{Find a vector} \quad & x \\ \text{that maximizes} \quad & c^T x \\ \text{subject to} \quad & Ax \leq b \\ \text{and} \quad & x \geq 0 \end{aligned}$$

Here, the variables to be determined are the components of x. The variables c and b are provided vectors, and A is a given matrix. The notation c^T indicates that the coefficients of c are employed as a single-row matrix for the purpose of creating the matrix product. The objective function is the function whose value should be increased or decreased. The constraints that define a convex polytope over which the objective function is to be optimized are $Ax <= b$ and $x >= 0$. In this sense, two vectors are comparable if they have the same dimensions. It can be asserted that the first vector is less than or equal to the second vector if each element in the first is less than or equal to its corresponding entry in the second.

Linear programming is a valuable technique for solving a wide variety of problems that need to optimize available resources. It could be used in the manufacturing industry to assess the best way to distribute workers and machinery in order to keep operational costs to a minimum. It has the potential to be utilized in high-level business operations, specifically in the process of determining which things to sell and in what quantities in order to achieve the greatest amount of profit. It could also be utilized in the field of logistics, namely in the process of determining how best to allocate resources in order to complete a task in the shortest period of time possible.

Chapter 3

Datasets description

Marabou can be used to analyze and verify the behavior of neural networks trained on a wide variety of datasets. Some common datasets that are often used with Marabou include:

- MNIST : A dataset of grayscale images of handwritten digits, commonly used as a benchmark for image classification tasks.
- MNIST Fashion : A dataset of grayscale images of Zalando's clothing items, also commonly used as a benchmark for image classification tasks.
- CIFAR-10 : A dataset of small images (32x32 pixels) in 10 classes, often used for object recognition tasks.

Marabou can also be used to analyze and validate the behavior of neural networks trained on custom datasets using standard machine learning frameworks like TensorFlow. During my experiments with Marabou, I exclusively focused on the MNIST and MNIST Fashion datasets to verify my models. CIFAR-10 was only used in Chapter 5. That's why I will explain those datasets a bit more in this section.

3.1 MNIST

MNIST [21] is a widely used dataset for the handwritten digit classification task. It consists of a training set of 60,000 examples and a test set of 10,000 examples. Each example consists of a 28x28 grayscale image (meaning that each pixel is represented by a single value ranging from 0 (black) to 255 (white)) of a handwritten digit (0–9) and a label indicating which digit is written. The label distribution for this dataset is uniform for the training set as well as for the test set. It means that among 60.000 labeled training examples, we can observe 6.000 examples for each of

the possible labels. The same applies for the test set with 1.000 examples instead of 6.000 for each possible label.

It was developed in the late 1990s by Yann LeCun, Corinna Cortes, and Christopher Burges. It was created as a benchmark for training and evaluating machine learning models for image classification tasks. It's derived from a larger dataset called the National Institute of Standards and Technology (NIST) Special Database 19, which consists of a collection of handwritten digits that were collected from American Census Bureau employees and American high school students. The MNIST dataset was made by taking a small part of this larger dataset and preprocessing the images to make them easier to work with and more uniform for machine learning tasks.

The MNIST dataset is widely used as a benchmark for training and testing image classification models. It is often used as a starting point for testing machine learning models and developing new techniques, as it is relatively small enough to be easily trained and evaluated on a personal computer and straightforward, yet still challenging enough to require the use of advanced machine learning techniques. The dataset is ready-to-use, but in our case, we will perform a normalization of the pixel values.

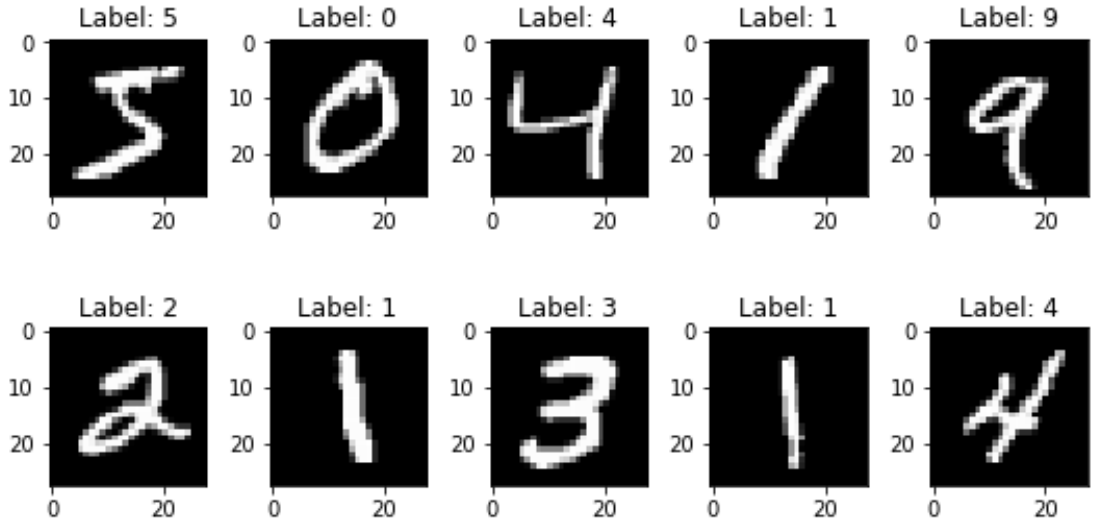

Figure 3.1: First 10 digits from the MNIST training set

Labels : [0,1,2,3,4,5,6,7,8,9]

3.2 MNIST Fashion

MNIST Fashion is a balanced dataset for the classification of Zalando's clothing items. When comparing machine learning techniques, Fashion-MNIST is a direct drop-in substitute for the original MNIST dataset. The image size and split structure between training and testing are the same (i.e., a training set of 60,000 examples and a test set of 10,000 examples, where each example consists of a 28x28 grayscale image and a label indicating the type of clothing item). It was made because the original dataset is nowadays considered too easy and overused and can not reflect modern real-life tasks. As a matter of fact, the MNIST Fashion dataset is a bit harder than the original one as the shapes of clothing are more complex and varied.

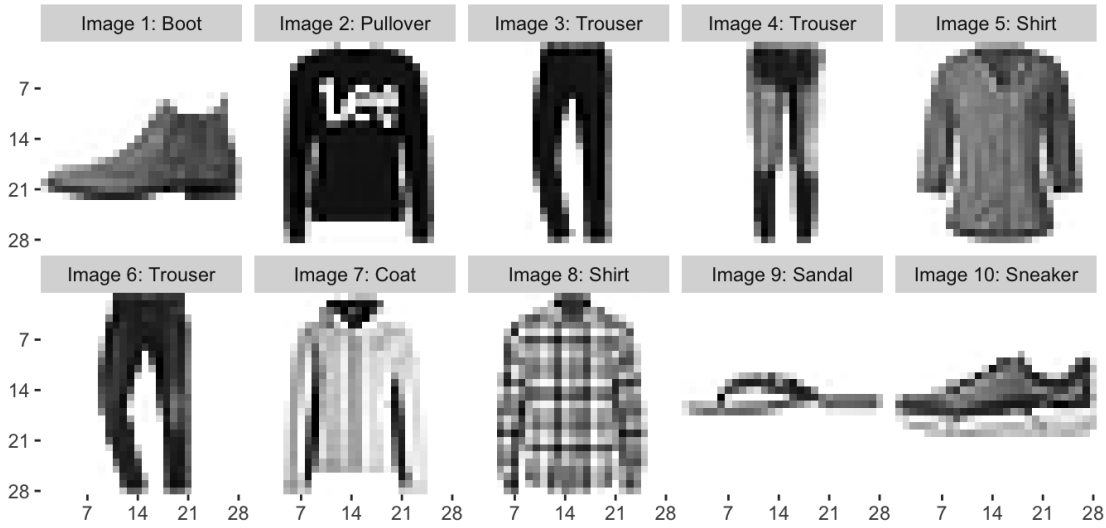

Figure 3.2: First 10 digits from the MNIST test set

Labels: [T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot]

3.3 CIFAR-10

CIFAR-10 (Canadian Institute for Advanced Research) is a balanced dataset for the classification of color images. It consists of a training set of 50.000 examples and a test set of 10.000 examples. Each example consists of a 32x32 color image and a label. The main differences with the previous dataset are that the training

set is smaller by 10.000 examples, the resolution is a bit higher (from 28x28 to 32x32), and the pixels are not grayscaled anymore, which means that the input size will be three times larger as it requires three color channels (red, green, and blue) rather than just one. This increases the difficulty of the dataset while keeping it within the range of binarized neural networks. In addition to that, both MNIST datasets contain only images with a homogeneous background, while the CIFAR-10 background contains various elements and is much more noisy.

Figure 3.3: Random 10 samples from the CIFAR-10 training set

Labels : [Airplane, Automobile, Bird, Cat, Deer, Dog, Frog, Horse, Ship, Truck]

Chapter 4

Models description

In my experiments, I will look at four different neural network models to see how their size affects their accuracy on a multi-class classification task when they are binarized. I will then use these models in my verification experiments. The models range in size from small to large and have been chosen based on their ability to learn complex relationships in the data and their efficiency in terms of computational resources. As the precise architecture of the binarized version has additional quantization in layers and an extra batch normalization layer after each input, dense, or convolutional layer, I decided to depict here a high-level overview for both model types::

1. Small model:
    - Input layer with 784 neurons
    - 1 dense layer with 200 units
    - Output layer with 10 neurons
2. Medium model:
    - Input layer with 784 neurons
    - 3 dense layers: 400 units, 200 units and 100 units
    - Output layer with 10 neurons
3. Large model:
    - Input layer with 784 neurons
    - 4 dense layers: 512 units, 512 units and 100 units
    - Output layer with 10 neurons

4. CNN model:

    - Convolutional input layer with with input shape (28,28,1), 32 filters, kernel size of 4 and strides set to 2
    - Convolutional layer with 64 filters, kernel size of 4 and strides set to 2
    - Flatten layer
    - 4 dense layers: 512 units and 256 units
    - Output layer with 10 neurons

All the models have a softmax activation function for the output. For all of them, I used the Adam optimizer as it is computationally efficient and tends to converge faster, the sparse categorical cross-entropy loss function as it can handle a large number of classes and computes the loss efficiently, and the accuracy as the metrics during the training of the models. All the models are trained with 5 epochs and a batch size of 64. As a reminder, the MNIST training dataset consists of 60.000 samples, and the test set consists of 10.000 samples.

The only exception will be made when doing experiments on the CIFAR-10 dataset, for which we will use the BinaryNet model trained as described in the CIFAR-10 Larq tutorial because previously described models are too simple for that dataset, achieving at most 40% accuracy. Moreover, the CNN model and the BinaryNet model will not be used in the Marabou verification experiments (only for binarized neural network evaluation).

Chapter 5

Larq Models

As explained in Chapter 2, I will be using the Larq library to easily build Keras-based binarized models. We will be conducting a series of experiments to evaluate the performance of binarized neural networks on the datasets described in Chapter 3 and compare them to traditional deep neural networks. We will also be examining the effect of different model architectures on the accuracy and uncertainty of binarized neural networks. Overall, our goal is to gain a better understanding of the strengths and limitations of binarized neural networks and how they can be applied in practical settings.

Make sure to set the weights and activations to binary before saving[22], and that models are saved in a Tensorflow protobuf format[23] and not in ".h5" format, as Marabou isn't compatible with the second one. Retrieving binary weights is especially important because it's not mentioned in the official LARQ binarized neural networks tutorials[7]. Without it, latent weights will be saved, which will make it hard for Marabou to do a correct verification later on.

	# Params (M)		Model size (KiB)		Training time (s)		Test Acc. (%)	
	DNN	BNN	DNN	BNN	DNN	BNN	DNN	BNN
Small	0.16	0.16	621	22	7.43	11.7	94.38	94.05
Medium	0.42	0.42	1618	59	13.37	21.64	96.31	96.14
Large	0.72	0.72	2796	101	19.98	30.11	97.09	97.07
CNN	1.77	1.77	6922	218	73.5	104.7	98.85	97.98

Table 5.1: Models comparison between the DNN and BNN version

As my first experiment, I decided to quickly evaluate the basic differences between the deep and the binarized versions of the models. As explained before, the only difference in the layers of both is that the layers of binarized neural networks have extra batch normalization layers and quantization is performed. The batch normalization layers are really important in the architecture, as without them the model won't be trained appropriately and the accuracy will drop drastically. We can observe in the first two columns of Table 5.1 that, as expected, the number of parameters is exactly the same for both types of models. This is not surprising, as the layers are supposed to be the same size. On the other hand, we can see a huge optimization of the model size, which is a direct outcome of the binarization. The model uses 1 bit to store the weight values, which can be either 1 or -1, instead of 32-bit double precision values like in the deep neural networks. The activation functions that the network uses are also quantized, resulting in outputs with fewer bits. As a result, the memory footprint of the network is greatly reduced, and its evaluation is both quicker and less expensive. We can expect a reduction factor of more or less 32, which is the case in our results. Even though the computation is supposed to be quicker for the evaluation, we can see that the training time is a bit longer for the binarized neural networks. Unfortunately, binarized neural networks often demand additional computations due to the necessity to approximate the posterior distribution over network weights. This can be resolved by building an optimized setup, such as a binary matrix multiplication graphics processing unit kernel [2]. As last, we can observe that binarized neural networks are able to achieve almost the same accuracy as deep neural networks. For all the models, the accuracy doesn't drop more than 1% which is really impressive taking into account how much smaller the model size is. We can see that, as in deep neural networks, models including convolutional layers are able to achieve greater accuracy, but in the case of binarized neural networks, the cost incurred by the model size increase is really negligible.

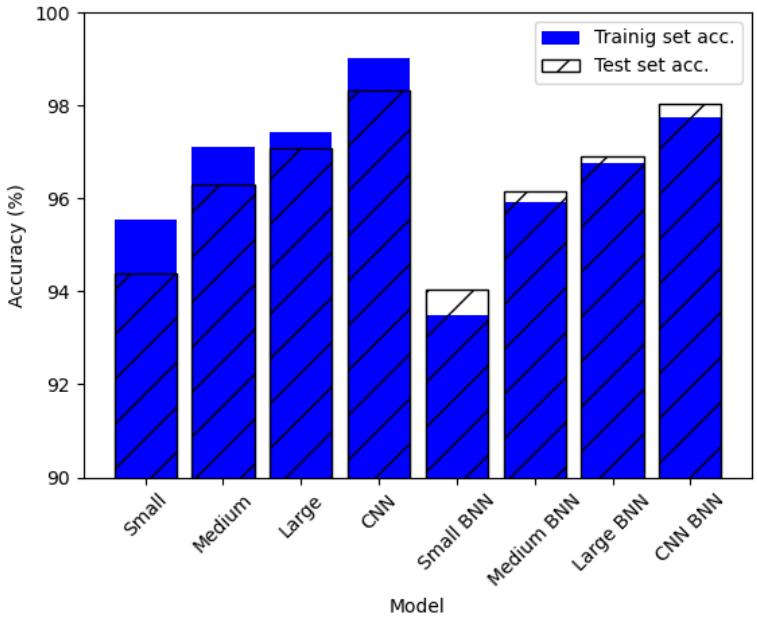

Figure 5.1: Training/test set accuracy for DNN and BNN models without normalization of input values

It is interesting to compare the average training and test set accuracies of the models on the MNIST dataset. Figure 5.1 shows us the test and training set accuracy comparisons for both model types. The average is a mean over 10 runs, which I considered to be representative enough. The test set accuracy scores are similar enough to what we obtained in Table 5.1, but an interesting property appears when we compare them to the training set accuracies. It appears that for the MNIST dataset, while a deep neural network will have a tendency to overfit the data, the binarized neural networks will trend in the opposite direction. This is also due to the binarization of values, as deep neural networks with floating point values will be able to better fit the training set, while binarized neural networks will more generalize due to the limited representation capacity as the values cannot reach the same degree of precision. This makes them less sensitive to the noise and variations in the training data. One possible explanation is the simplicity of the dataset. The MNIST dataset is often considered too trivial for today's cutting edge. Thus, we will also perform experiments on the MNIST Fashion and the CIFAR-10 to see if it's a usual behavior.

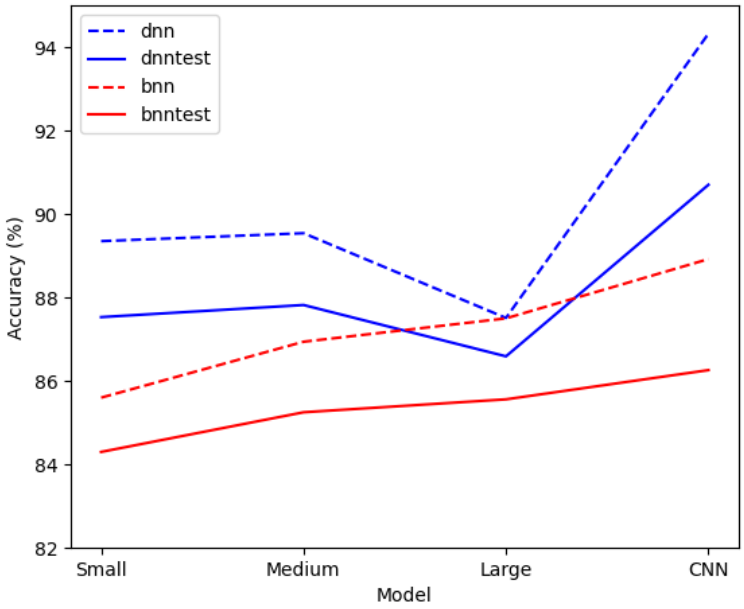

Figure 5.2: Overfitting comparison on the MNIST Fashion dataset for both model types

As we can observe in Figure 5.2, it's enough to replace the MNIST dataset with the MNIST Fashion dataset to see the underfitting behavior disappear. As a matter of fact, even though the MNIST Fashion dataset has samples with the same resolution, the labels to classify are slightly more difficult as they represent images of clothing and accessories, which are more complex and diverse than images of handwritten digits. Furthermore, the difference in the test set accuracy score between the deep neural network and its binarized version grows increasingly significant (up to 6% for the MNIST Fashion versus 1% in the MNIST dataset), implying that the same architectures used for deep neural networks are no longer sufficient for binarized neural networks to follow the accuracy score and that more complex architectures are required to achieve comparable scores. But it isn't an issue, as we saw that we have some margin thanks to the model size reduction. Nevertheless, we will keep those architectures for further experiments, including the MNIST Fashion dataset. We will just consider those models weaker and keep that in mind while analyzing the future results.

The pattern changes when we try experiments on the CIFAR-10 dataset. I trained the model using Google Colab GPU [24], and after 1 hour of training I could achieve a test accuracy of around 83% while the training accuracy was more than 99.5%, which is a huge overfitting but it seems necessary for the binarized

neural networks to achieve good results. We can find examples of deep neural network models [25] with test accuracy of 86% and training accuracy of 89% on the internet. That proves that a deep neural network model is able to achieve even better results without overfitting that much. This shows clearly a limitation of the accuracy that binarized neural networks could eventually achieve on the CIFAR-10 dataset due to the dataset's complexity. Although some research has been done to tackle that limitation by finding some specific architectures that are better suited for binarized neural networks [26, 27], like custom convolutional layers, etc. Those solutions are proving that there is still a lot of room for improvement in that matter, and we are only in the early stages. For the moment, we are far from being able to take 100% advantage of the capabilities of binarized neural networks, especially when it comes to their use in real life complex situations.

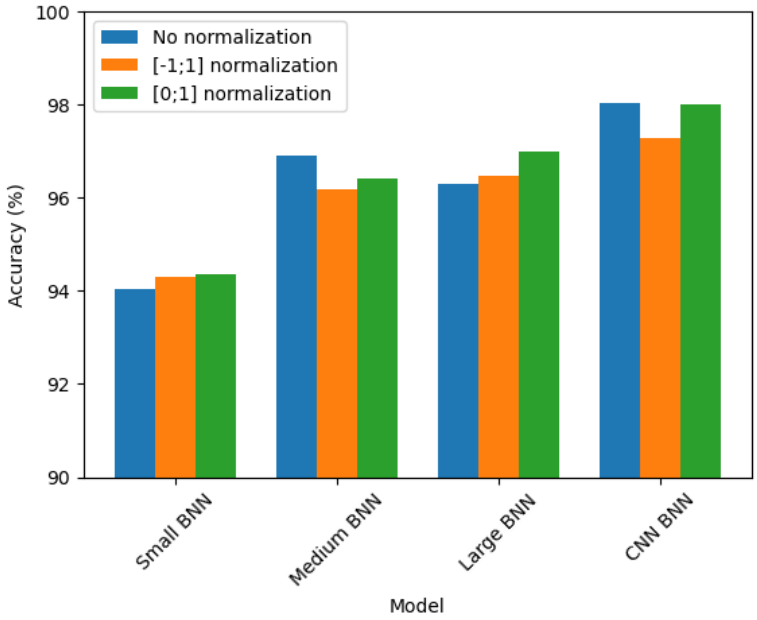

Figure 5.3: Test set accuracy depending on input normalization

The raw MNIST dataset is composed of grayscale images with pixel values in a range from 0 to 255. As mentioned in Section 3.1, we will perform a normalization of those values as it is usually a good practice before training the models. Even in the Larq tutorials, they suggest normalizing the values in a range between -1 and 1, as this gives the most accurate results. On Figure 5.3, we can see the different results that I reached with playing with that parameter, and we can see that, for the small and large models, the normalization slightly improved the results.

Moreover, I tried to normalize the values between 0 and 1 instead of -1 and 1, and, in general, I was able to improve the accuracy even more, especially for the CNN model, for which the normalization between -1 and 1 wasn't working so well.

Without any doubt, we can say that binarized neural networks, thanks to the simplification of the computations, are able to achieve results almost as good as those of deep neural networks and, at the same time, considerably reduce the model sizes. Obviously, in most cases, the model architecture will be bigger as they need more layers and units, but that difference isn't impacting the model size in an apparent way. Even the biggest model for our experiments is only 1.26 MiB large, which allows it to be run on smaller devices and produce really satisfying results.

Chapter 6

Marabou methodology

In this section, I will describe, step by step, how Marabou proceeds with a model verification and explain all the components of such a procedure. I will also address different problems that Marabou encounters in order to verify our models.

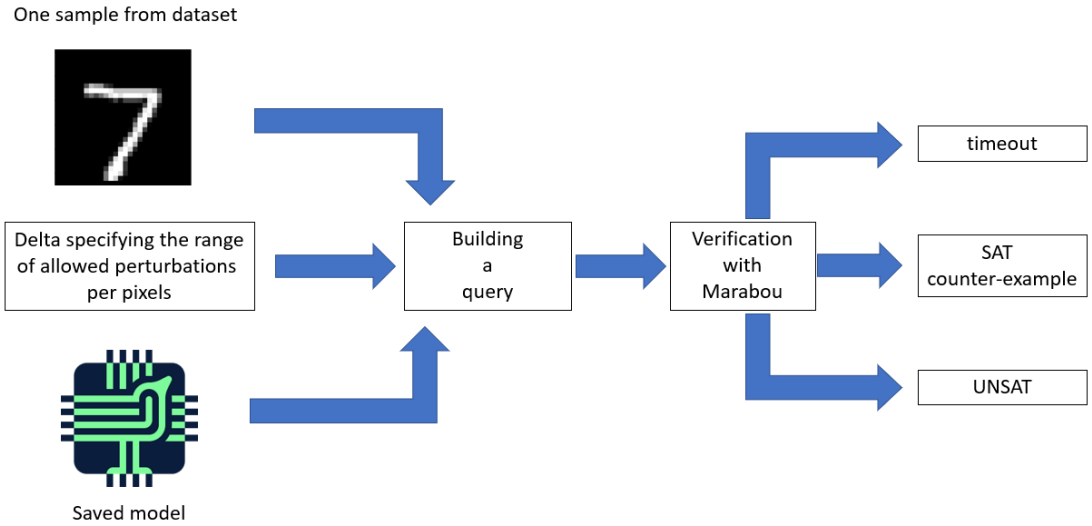

Figure 6.1: Marabou procedure

Figure 6.1 illustrates the procedure for Marabou verification. We will need 3 things at the beginning: a LARQ binarized neural network model, 1 sample from the dataset on which we want to perform the verification, and a delta specifying the amount of perturbation that is allowed per pixel. Those 3 elements will be used to create an input query, which will be fed to Marabou. Marabou will perform the verification and will output for us one of the three possible outcomes: a SAT assignment, an UNSAT, or a timeout declaration.

6.1 Before verification

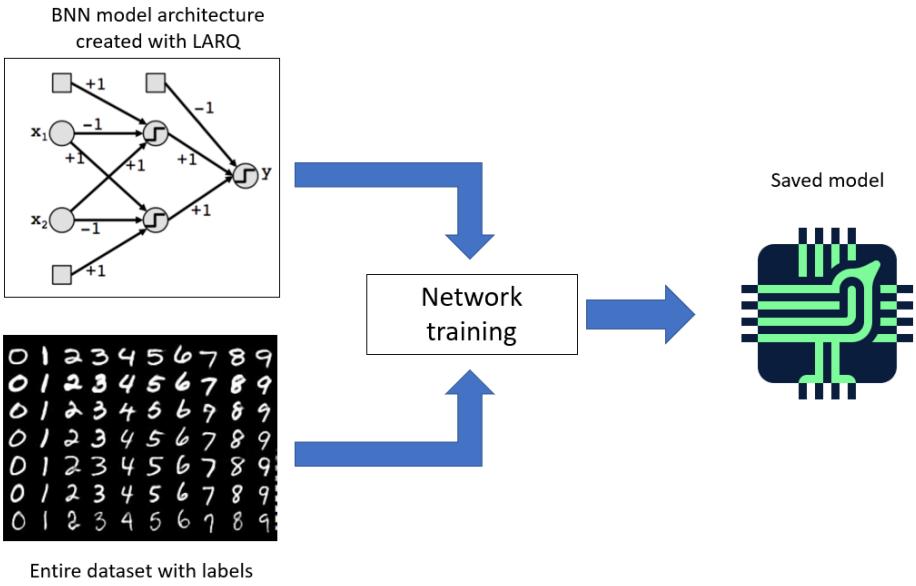

Figure 6.2: Training a binarized neural network with LARQ on the MNIST dataset

First of all, as described in Figure 6.2, we need to create a model with a specific architecture, including quantized layers from LARQ and batch normalization layers from Keras. Then we train that model on a large labeled dataset of our choice and evaluate its accuracy to see how well the model is performing. Then we save the binary weights and the model structure in a protobuf file.

As we saw in Figure 6.1, the verification will be performed on only one sample from the test set. Intuitively, if we want to figure out how robust a model is, we should test it on the whole dataset. However, testing a model on the whole dataset can be computationally expensive and take many days; some tests can even take more than 2 hours.

The last initial parameter is the delta parameter. It is a parameter that defines the amount of perturbation allowed per pixel. Caution has to be taken when setting that parameter, as there may be some confusion about it. In fact, Marabou uses two types of units to define the perturbation: delta and epsilon. Following that simple rule, they represent the same amount:

$$\% \text{ of perturbation allowed per pixel} = \text{epsilon} = \text{delta}/256$$

5% of perturbation is equal to an epsilon of 0.05 and a delta of 12.8. What does the amount of perturbation represent, and why two different units? As an example, if we take the delta from the example above (i.e., 5%), we will allow Marabou to perform modifications on each pixel of the image up to +/- 5%. If we have a pixel with a value of 0.70, Marabou will be allowed to test any values for that pixel between 0.65 and 0.75 to see if it produces a counter-example in combination with the rest of the pixels (modified or not). We have 2 different units because delta is used when we create the query, which uses raw examples (pixel values between 0 and 256), and epsilon is used during the verification when the image is normalized (pixel values between 0 and 1). It goes without saying that the higher the delta, the greater the allowed perturbations, and thus the larger the search space. However, it does not necessarily take longer to find a SAT assignment because the probability of finding an adversarial example increases with a larger perturbations range. Trivial example: if we set the delta to maximum (i.e., 256), the solver will instantly find an adversarial example, which would just be an image with identical values for each pixel. In general, a smaller value will lead to stricter verification, while a bigger number will give more freedom in the perturbations taken into account.

To run the solver, Marabou offers 3 different possibilities:

1. By providing a network and a property file.
2. By providing a network, a dataset (–dataset), an epsilon (-e), a target label (-t), and the index of the point in the test set (-i).
3. By providing an input query file (–input-query).

The first method isn't appropriate for our case, as we are not trying to verify a property of a model but its robustness. The second method isn't appropriate for binarized neural networks created with Larq for multi-classification as we use a softmax activation function, which isn't implemented in Marabou. We will take a look at how to solve that issue in the section 6.5 using the last option. As I just said, we will run the solver for my experiments by providing an input query. Marabou itself doesn't provide any indication on how to build such queries; that's why I needed to use the scripts provided to me by the authors of the extension paper [6] that I adapted afterwards. Building such a query is complicated as it's a big file that contains all the information about the initial inputs: the number of layers, number of inputs, number of outputs, layer sizes, various flags, input lower and upper bounds, means and range of the inputs (normalization), the sequence of weight matrices and bias for each layer, etc. Thus, the scripts were a great help. From the start, the script takes the three elements as input and checks that the model correctly classifies the sample we provided. It doesn't make sense to check

the robustness of a sample for which the model is already wrong. Then, if it's well classified, it will look at the score distribution for the different possible labels. With the best score attributed to the correct label, Marabou will run up to the second-best score label, as explained later in Section 7.1. It will compute all the information mentioned before and produce an input-query file.

6.2 Verification

Once the query is created, we will feed it to the Marabou solver. Here is how Marabou defines the problem: suppose a binarized neural network, called B, that converts an input vector x into an output vector $y = B(x)$, a pre-condition $P(x)$ and a post-condition $Q(y)$, it will attempt to ascertain whether an actual input x_0 exists such that $P(x_0) \wedge Q(B(x_0))$. P and Q are, respectively, conjunctions of linear restrictions over the input and output neurons, and B, typically, indicates an unwanted output of the binarized neural network; therefore, the presence of such an x_0 is indeed a counterexample.

Marabou extends the Reluplex algorithm [28] in which the node values of the neural network are viewed as variables and the verification query is viewed as a set of constraints on these variables. The solver's purpose is to discover a node assignment that satisfies P and Q. There are 2 kinds of constraints: linear constraints (easy to solve) and piecewise linear constraints (difficult, making the problem NP-complete). At first, an iterative version of the Simplex algorithm [29] is used to find solutions to the linear constraints. As soon as the linear constraints are met, it makes incremental changes to the assignment in an effort to fix any broken piecewise-linear constraints. The linear constraints are again addressed if these procedures led to violations of them. This procedure converges frequently.

6.3 Results

As said in Section 6.2, the verification usually converges. But unfortunately, it can be quite slow, hence the need to set a timeout, which is discussed in Section 6.4. If a verification exceeds a certain time, it will stop and return as a result of a timeout. On the other hand, if the solver finds a solution in time, it can either output UNSAT when there are no adversarial examples found or SAT when it discovers an adversarial example, in which case it will also output the variables assigned that produce that result.

6.4 Timeout

As discussed in Section 6.3, a timeout is inevitable, so I will discuss it in this section. As we will see in Table 7.2, even when the model is pretty small and the epsilon doesn't allow a wide perturbation range, Marabou can take a long time looking for a SAT when such an assignment exists or exploring the whole search space in order to prove that such an assignment doesn't exist. As the Marabou extension for binarized neural networks is designed for small and medium models, we will quickly be faced with queries that can take a very long time to prove a SAT or UNSAT assignment, either because the model became too big or the search space was too vast due to the high precision. Even though Marabou has tools to speed up SAT/UNSAT assignment, like different heuristic searches, warm-start, the Split-and-Conquer strategy, lowering the minimum threshold to fix a constraint, etc., queries that can't finish in a reasonable amount of time are still a problem. As a result, during my subsequent experiments, I decided to set a timeout of 2 minutes and 30 seconds in general. If Marabou cannot solve a query within that time, it will be considered a timeout. The main issue with that approach is that if a SAT assignment exists but requires a lot of searching, it will never be found. But it's a common, if not perfect, solution in the current literature to those kinds of situations. Even more so since the extension paper shows that some questions can't be answered in 2 hours. During this thesis, I liked to figure out how stable a model was by doing shorter experiments on a large number of samples where larger changes were allowed. It seemed more relevant for me and was more doable within the framework of my thesis. As the matter of time seems so crucial, in Section 7.2, we will measure the impact of improvements brought by the Gurobi solver.

6.5 Bypassing the impossibility of implementing the softmax function

As said before, Marabou is a constraint verification tool, and therefore it's impossible to implement the traditional softmax function as we would do it in other frameworks working with neural networks. That's why I needed to use a bypass to be able to verify my models. The bypass will be made when creating the query and will imply a limitation for the framework. So, step by step, in order to run my script for the query creation, I needed to provide a model, a delta, the number of samples I wanted to create, and, most importantly, the output of the layer just before the softmax layer. This is important, as Marabou won't recognize the softmax layers and throw an error. We specify in the script the index of the sample we want to verify our model on, after which the script will check if, before anything

else, our model classifies this sample well. If not, it will just take the next one. However, if our model correctly classifies, it will proceed to query creation. And here comes the trick. Let's say we work with a sample labeled "9". The constraint that should verify later on that we indeed have found a SAT example should look somewhat similar to this:

$$y_9 \leq y_0 \mid y_9 \leq y_1 \mid y_9 \leq y_2 \mid y_9 \leq y_3 \mid y_9 \leq y_4 \mid y_9 \leq y_5 \mid y_9 \leq y_6 \mid y_9 \leq y_7 \mid y_9 \leq y_8$$

y_i being the activation value that the model gives for the output to be predicted as the label i. The constraint basically just looks for any output activation value to be higher or equal than the one for the correct class. Unfortunately, such constraints are impossible to formalize in constraint programming implemented in Marabou; that's why we need to do a simplification that consists of taking the second-best prediction of the model at the beginning and making it the only label Marabou will be running up to. So, my script will take the second-best label and put it as the final constraint to verify the SAT solution. Let's suppose the second-best output is 8, and our constraint will look like this: $y_9 \leq y_8$.

It is clearly a limitation in terms of verification as, now, Marabou will try to find different combinations of perturbations that will cause our model to misclassify the example only as an eight and will not take into account all the other possibilities, which could be a SAT solution as well. But as far as I could tell, this is the only acceptable solution for the softmax layer.

In general, after investigating the documentation and going through the available literature, I don't think that Marabou supports multi-class classification as is. One less desirable alternative would be to run the solver with a different label as the target each time and then redefine the robustness assessment as either the quickest time between the runs to reach a SAT or the mean time for all the labels, but in both cases that would in theory multiply the solving time by 10. In addition to that, in the second case, the results would be biased as we would take into account all the searches. Consider an example in which the solver instantly finds a counter-example with a specific target but is UNSAT for all other labels. How should we consider this example? Is the model rather robust or not at all?

Another solution would be to modify the initial models and instead of having multi-class classifiers, train a binary classifier using the "one-versus-all" strategy, where the output would be either "correctly classified" when the model predicts the correct class or "misclassified". But here again, we would have the softmax function problem when computing the "misclassified" probability. If we opt for another

function, we will not only probably lose a lot in accuracy but also in interpretability, which is absolutely not desirable.

6.6 Fixing some pixel regions before the verification

An interesting aspect of verification is the ability to see what adversarial examples cause our model to be tricked. Thanks to that, we can analyze some specific common characteristics between them and take them into account. During my experiments, which I will explain in Section 7.3, I observed that the solver is sometimes able to find counter-examples by applying perturbations only to some regions of the image. I investigated how the solver behaves when we limit its ability to apply perturbation to specific pixels rather than allowing it to apply perturbation to the entire image[30, 31]. Such an approach may be useful to verify the model with respect to perturbations that preserve certain properties of the input, such as the overall shape or bounds. This could be useful in applications where it is important to maintain the integrity of certain features of the input even in the presence of perturbations. In my experiments, I was able to implement a thresholding function that fixed a pixel when it met the condition (i.e., when a pixel's initial value is greater or equal to 0.5). Another option is to let users specify preferences or weights for certain perturbations. For example, they could penalize perturbations that alter the input's general structure more severely than those that just modify minor aspects. Users might then be able to compromise between the degree of verification rigor and the latitude given to perturbations. But I didn't implement that approach in my experiments.

Chapter 7

Marabou and verification

In this chapter, we will proceed to the verification of the models described in Chapter 4. We'll see if the method we used to overcome the constraint programming limitation for the softmax implementation is viable, we'll analyze the Gurobi solver's performance and discover how Marabou behaves with different deltas, and we'll try to improve solver efficiency even more by analyzing the behavior when we allow it to perturb only some pixels. Note that:

- Marabou no longer provides Windows support, and you might run into some issues if you try to work with the most recent version of the framework
- Some modifications might be needed before building the framework on the machine. I described the ones I had to perform in Section 10.1.
- We will perform experiments only on the MNIST and MNIST Fashion datasets as our basic models reach too low of a accuracy on the CIFAR-10 dataset for the verification to be relevant
- CNN models won't be verified in this section as LARQ is fusing the ReLu and BatchNormalization operations into the convolution operation [32]. This creates a new type of operation which is not supported yet by Marabou. The only way to bypass that is to put a batch normalization layer after the flatten layer but that allows us to use only one convolutional layer which gave me poor results.

7.1 Impact of bypassing the softmax implementation

As we discussed in Section 6.5, the Marabou framework faces an issue with multi-class implementation for the softmax layer. In this section, we will try to see the

impact that it can have on the results and how reliable our bypass is.

Label	Initial score	Result	Time(s)
0	-18	UNSAT	/
1	-28	UNSAT	/
2	-12	UNSAT	/
3	-20	UNSAT	4
4	6	UNSAT	4
5	-10	UNSAT	2
6	-8	UNSAT	63
7	-16	UNSAT	/
8	26	**SAT**	42
9	42	/	/

Table 7.1: Results of a query with the small model and a delta of 2.2 on the sample 62 (label 9) according to the label the verification is running up against

Label	Initial score	Result	Time(s)
0	4	**SAT**	482
1	-2	UNSAT	110
2	-26	UNSAT	/
3	-2	UNSAT	31
4	0	UNSAT	352
5	32	/	/
6	-10	UNSAT	53
7	-18	UNSAT	4
8	24	**SAT**	6
9	-12	UNSAT	20

Table 7.2: Results of a query with the small model and a delta of 2.2 on the sample 52 (label 5) according to the label the verification is running up against

After all, as we can see on the Tables 7.1 and 7.2, the limitation mentioned before isn't so problematic as, in general, most of the adversarial examples that can be found are precisely labeled as the second-best output of the initial sample. We can see in Table 7.1 that the only query for which Marabou could find a SAT solution is the label with the second highest initial score. It doesn't mean that we will never be able to find a SAT solution for any other label. Indeed, we can see in Table 7.2 that for the sample 52, which represents a 5, Marabou could find a SAT assignment for the labels that have the second and third best initial scores. Nevertheless, we can see a huge time difference between the solutions. When the runner-up label is the second-best, it could find a SAT solution in barely 6 seconds, while for the third-best solution, we had to wait almost 6 minutes.

7.2 Gurobi

As explained before, Marabou can struggle when the query becomes too large, and that's why we will see how we can improve the solving time with the help of the Gurobi optimizer.

Model	Number of equations	Number of vars before pre.	Number of vars after pre.
Small	411	1794	1330
Medium	1411	4294	2779
Large	2259	6414	4023
CNN	11563	37786	18648

Table 7.3: The evolution on average of the number of equation, the number of variables before preprocessing and after preprocessing for a delta of 10.0 for each model

Model	MNIST		FASHION	
	delta	time (s)	delta	time (s)
Small	5	1	7	1
Medium	0.08	3	0.4	3
Large	0.07	8	0.2	4

Table 7.4: Comparison of the minimal delta values to declare as quickly as possible, after preprocessing, that the query is UNSAT on one sample

As we can see in Table 7.3, the number of equations grows accordingly to the model size, and the number of variables, even after preprocessing, can quickly become high, especially considering that the number of variables eliminated by the preprocessing decreases as the delta increases. In Table 7.3, for example, we used a delta of 10 (epsilon of 0.04), which is often thought to be the lowest delta needed to make changes that can be seen by the human eye. However, we could easily use a higher delta and increase the number of variables after preprocessing.

In table 7.4, I tried to see the impact of the accuracy of a model on the ability for the solver to find an UNSAT after the preprocessing. But the three models don't behave the same way for a single sample, and the Fashion dataset is completely different. That's why, instead of running the solver on the same samples of each dataset, I tried to find samples for which the models have a score higher than 90 for the correct label and the same difference between the correct label and the label it's running up to. So, regardless of the model or dataset, the examples provided to the solver represent the same difficulty. As is to be expected, the more complex the model, the more difficult it is for Marabou to quickly find an UNSAT for the MNIST dataset. The larger the model, the more we have to decrease the delta to quickly find an UNSAT, as decreasing the delta reduces the search space and facilitates solving for Marabou. As we can see, that stays true for the MNIST Fashion dataset, but as we saw in Chapter 5, the accuracy of the same models is

much lower for that dataset. We can observe that it helps Marabou in finding an UNSAT, as for all the models, we could increase the epsilon without increasing the searching time. We can clearly state that Marabou struggles more when the model is complex and the accuracy is high, which is a good sign and could be interpreted as an indication of robustness. However, there is still the issue of queries taking too long to return an SAT or UNSAT result when the epsilon is greater than those minimal values, necessitating a call to Gurobi for assistance in optimizing the search.

Label	Native				Gurobi			
	sat time(s)	sat num.	unsat num.	timeout	sat time(s)	sat num.	unsat num.	timeout
0	/	0	4	6	/	0	4	6
1	/	0	0	10	/	0	0	10
2	4	5	0	5	1	4	4	2
3	8	2	0	8	38	5	0	5
4	6	6	0	4	5	7	0	3
5	3	4	0	6	1	6	0	4
6	12	3	0	7	18	4	0	6
7	26	6	0	4	19	8	0	2
8	/	0	8	2	/	0	8	2
9	/	0	0	10	14	3	0	7
MEAN	9.8	2.6	1.2	6.2	13.7	3.7	1.6	4.7

Table 7.5: Comparison of the average on 10 samples of the time to find a SAT assignement, the average number of SATs, UNSATs and timeouts between native solver and Gurobi for each label with the small model and an delta of 5

In Table 7.5, we can see, with the small model, that the average time to find a SAT assignment increases when we use Gurobi, which is surprising at first. This occurs because Gurobi can find more solutions, particularly those that the native solver could not find before the timeout, which are not taken into account in the mean. We can observe that the average number of timeouts decreases, and also that in both cases, the solver seems to show that the model is more robust for label 8 compared to the other ones.

Label	Native				Gurobi			
	unsat time(s)	unsat num.	instant unsat	timeout	unsat time(s)	unsat num.	instant unsat	timeout
0	105	3	4	3	27	4	4	2
1	109	4	4	2	4	4	4	2
2	54	3	4	3	18	5	4	1
3	137	2	4	4	35	3	4	3
4	/	0	8	2	98	2	8	0
5	99	2	3	5	5	3	3	4
6	18	2	6	2	3	4	6	0
7	9	7	2	1	11	8	2	0
8	38	4	6	0	3	4	6	0
9	53	3	3	4	6	5	3	2
MEAN	69	3	4.4	2.6	21	4.2	4.4	1.4

Table 7.6: Comparison of the average on 10 samples of the time to declare UNSAT, the average number of UNSATs, instant UNSATs (found throughout preprocessing) and timeouts between native solver and Gurobi for each label with the medium model and a delta of 0.2

On the other hand, in Table 7.6, I've decided to evaluate the mean time for the medium model to find an UNSAT assignment because the model is more accurate and could struggle to find a SAT assignment on time. With the help of Gurobi, not only were we able to decrease significantly the time it took to find that an example was UNSAT and the number of examples that timed out, but we could also find solutions for labels for which the native solver couldn't find any (i.e., label 4). Moreover, we were able to confirm what we found in Table 7.5: that the models, in general, are more robust when we try to find adversary examples for the label 8, as we always found that the examples were UNSATs. Notice that the UNSATs found instantly are exactly the same for both solvers, as they are using the same preprocessing provided by Marabou. Since that shows that Marabou works better overall with Gurobi, I will use the Gurobi extension for the rest of my tests.

During my experiments, an attempt was made to define the robustness of a model not by comparing the time to find a SAT assignment but, in the case when such an assignment can be found, to count the number of possible SAT cases, such as those in this paper [33], and compare it to the total number of perturbations allowed in this specific query. That approach could allow us to estimate a model's robustness in a more pertinent manner. Unfortunately, the main bottleneck of that approach is the average time for a SAT query, which is the same bottleneck as

described in Section 6.4. Not only is that approach not scalable to bigger queries, but, as a matter of fact, it is not possible to easily find all the other solutions using the native solver or Gurobi, as in our case it's a linear programming problem. In fact, linear programming requires really specific manual algorithms or various heuristics to find multiple solution points. In addition to that, our models are numerically very challenging, which can cause the linear programming problem to have countless solutions. Some solutions are possible, such as focusing on the computation of all the vertices of the optimal facet, but they turn out to be very expensive.

7.3 Adversarial attack by region

One interesting aspect of neural network verification, as explained in Section 6.6, is to see what sort of perturbation usually leads to an adversarial example. Could one random pixel influence the whole evaluation? Are there common points between the different perturbations? How do counter-examples vary according to the epsilon allowed? What happens when we fix an area of pixels?

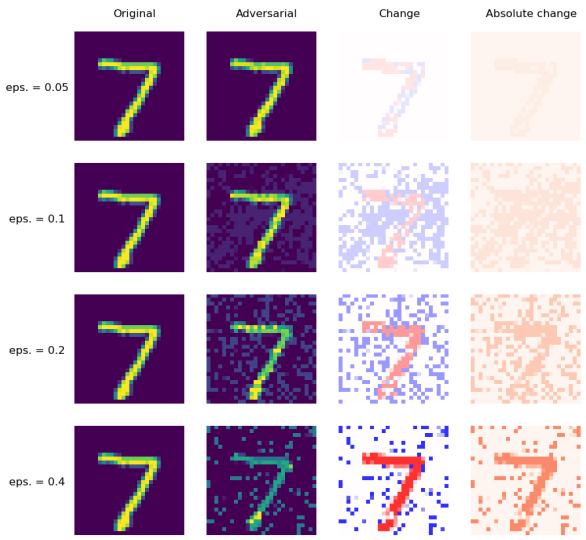

Figure 7.1: Evolution of perturbations used to found an adversary example according to the epsilon value

In Figure 7.1, the first column represents the image of the original sample, the second one is the adversarial example found, the third represents the increases

(blue) and decreases (red) done to the sample, and in the last column we can see the absolute intensity of the perturbation. By doing experiments, I observed an interesting behavior of Marabou. The further we go into the extreme epsilon values (allowing very small or very large perturbations), the more Marabou focuses on the pixels corresponding to the sample label to find an example. With intermediate epsilons, on the other hand, it applies perturbations to the entire image. I implemented some scripts to fix some pixels and see what effect they have on the modifications performed by Marabou and the impact that they will have on the time.

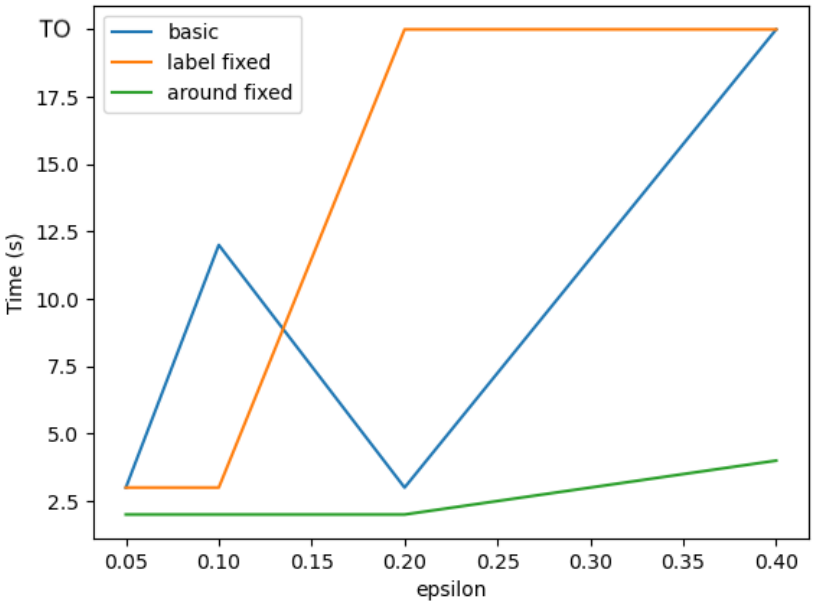

Figure 7.2: Time to find a SAT assignment when fixing different regions on the test sample 1194 (label 7)

In Figure 7.2, we can see the comparison of the time to find a SAT assignment by the medium-sized model for different epsilon values. The timeout is set to 20 seconds this time. The blue line corresponds to a pure basic query, the yellow line corresponds to a query where we fix the pixels corresponding to the label (pixel $>$ 0.5), and the green line corresponds to a query where we fix all the pixels around the label (pixel $<$ 0.5). It shows some really surprising results. For an epsilon of 0.05, fixing the pixels of the label doesn't seem to bother Marabou at all, even though we saw that with this epsilon, Marabou is able to find a SAT by focusing on perturbing those pixels. On the other hand, helping Marabou focus on the

label pixels by fixing the pixels around improves the result by 1 second, which is an expected behavior. For an epsilon of 0.1, we saw that Marabou tends to perturb any kind of pixel to find a SAT assignment, but the figure proves that if we would reduce the number of variables in the search by fixing some values, it would improve the Marabou performance. We would expect the same behavior with an epsilon of 0.2 as the kinds of initial perturbations to find a SAT are very similar. Nevertheless, we can see that when we fix the pixels of the label, the query times out. This would let us suppose that for an epsilon of 0.2, Marabou needs to alter both kinds of pixels to find a counter-example. The results for an epsilon of 0.4 are more understandable. As we saw, the higher the epsilon, the more Marabou will focus on the pixel corresponding to the label. It is hardly a surprise to see that when we help by fixing values that do not correspond to the label, Marabou is able to find a SAT assignment, whereas before it timed out for the two other queries. Overall, we can see that the most effective way to improve Marabou performances is by fixing the pixels around the label itself (which, in the case of MNIST, is rather easy as a thresholding condition on the pixels is enough to identify them, but for datasets such as CIFAR-10, we would need a more elaborated function as the samples are more complex). This can be explained firstly because it helps Marabou focus on the pixels that have the most impact on the prediction, and secondly because, by doing so, we reduce the search space and the number of variables, which can be really large with complex models. As a final observation, we can also see that models are much easier to trick when the perturbations being performed are very small and not visible to the human eye.

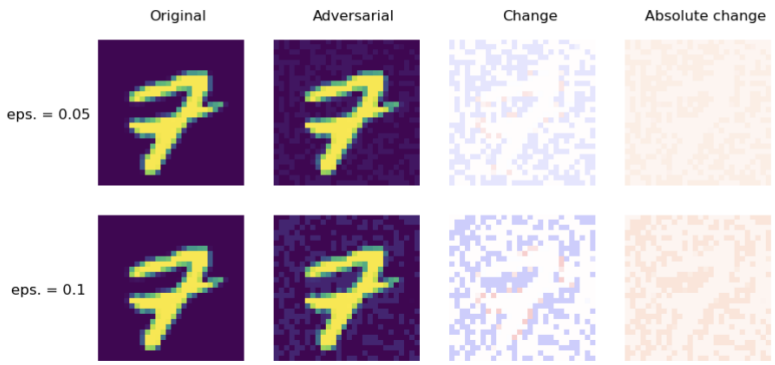

Figure 7.3: Evolution of perturbations used to found an adversary example according to the epsilon value when the pixels corresponding to the label are fixed

I investigated more in depth the visual results corresponding to the experiments done in Figure 7.2, and as stated, we can see the results only for the epsilon of 0.05 and 0.1. In Figure 7.3, we can clearly see that Marabou needs to modify a lot of pixels in order to reach an SAT assignment. That proves that when we perturb the

background, the model struggles more to perform a correct classification. Which partially proves the theory from Section 3.3 that the binarized neural networks encounter more difficulties when the images on which they are tested contain a more noisy background, such as CIFAR-10, than when the background is uniform and homogeneous, like in the MNIST dataset. The perturbations also need to cover a wide area, as I never found adversarial examples found by Marabou where only a few pixels were modified. As previously stated, this experiment makes an intriguing observation, namely that the model is easier to fool when the background noise is low. That is also the reason why we cannot quickly find an adversarial example with epsilon 0.2 and 0.4 when we fix the pixels corresponding to the label. Perturbations of such importance aren't able to trick the model when performed on the background, and the search space becomes too big, so the solver needs time to find an adversarial example with smaller perturbations, such as epsilon 0.1.

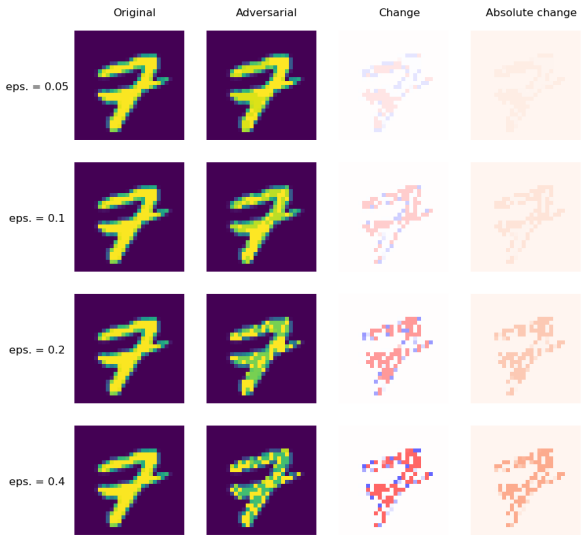

Figure 7.4: Evolution of perturbations used to found an adversary example according to the epsilon value when the pixels not corresponding to the label are fixed

By comparing the results in Figures 7.1 and 7.4 (especially for an epsilon of 0.4), we can see that in both the normal case and the case where we fixed the pixels, the model tends to be tricked when the label seems to be more and more merged into the background. We can see that Marabou increases the pixels on the borders of the label and decreases the more central ones so that they become more uniform.

Chapter 8

Conclusion

Neural networks allow us to do incredible things on a daily basis. However, they are not perfect and fail sometimes. In some cases, the failure is negligible, but there are contexts where failure can be critical, such as in human face recognition. Even if binarized neural networks are not yet ready to be used in such complex situations, their robustness remains important even for simpler tasks. This is where verification enters the picture. We showed here how even for the best models, there are adversarial examples that can be found using verification frameworks such as Marabou.

With this thesis, I demonstrated some intriguing behavior of binarized neural networks on the most popular deep learning datasets. We saw that they can achieve almost state-of-the-art results while reducing a lot of the model size and computation complexity and adding appropriate normalization. However, there are still some limitations when it comes to more complex datasets, but there has been more and more literature appearing lately that proposes solutions to tackle that issue. I also depicted the Marabou methodology, showed how we can build input queries for the binarized neural network models so that we bypass the limitation of Marabou for the traditional softmax function due to the constraint programming base on which the solver is built, investigated the necessity of a timeout in such queries, and evaluated the improvements brought by Gurobi. My study also contributed to exploring the potential trade-off between the complexity of a model, its accuracy, and its robustness. And as last, I exposed the impact of queries with fixed-value pixels on the modifications performed by Marabou and on its efficiency, and I discovered the perturbation patterns that are able to trick the models.

As is reflected throughout the entirety of my thesis, whether in relation to binarized neural networks or their verification, the domains are still in their early stages. One development path would be to continue the research about the binarized model optimization. Either by developing more specialized hardware for binary operations

or by delving deeper into various paths for optimizing the methods used. Many novel ideas could consist of building custom optimizers or exploring the impact of the architecture of the model. We saw that if we kept the same architectures, the models could not follow when tested on harsher examples, so the idea would be to find new architecture styles that could better fit the binarization.

Concerning the verification of such models, we saw that even if Marabou was adapted for binarized neural networks, it still struggled with more complex models that included convolutional layers merged with batch normalization layers. To be fully compatible with LARQ models, a special extension that would include new constraints encoding related to those operations, might be a good consideration for the future. The main disadvantage of Marabou is that it faces scalability problems, causing the verification of bigger models with better accuracy to take sometimes more than 2 hours for a sample. Thus forcing the use of a timeout. One avenue of investigation would be to see what effect reducing the precision degree of the pixel perturbations would have, as I believe it does not need to be so small (10^{-6}). An alternative would be to investigate another distance metric for defining adversarial examples, as for now most of the literature uses the L-p norm.

As we saw in the last section, some interesting further work might be done about developing methods for incorporating user-specified constraints or preferences into the verification process. For example, it may be useful to test how well a binarized neural network can handle changes to the input that don't change certain properties, like the overall shape or texture.

One thing that I would have wanted to explore but unfortunately didn't have the opportunity to do is investigate the binary neural network verification of a wider range of applications, such as real-life situations. A tinyML talk[34] was done about the ability of LARQ models to detect if a person is in the frame of the camera or not. It would be interesting to try out the verification in such situations.

As a final research avenue, we could consider implementing a generative adversarial network (GAN) for LARQ models and measuring its efficiency on robustness verification. How close to complete robustness can we get using such methods?

To conclude, I would like to say binarized neural network verification appears to be more efficient and sound than the one for the traditional deep neural networks, and this is the reason I believe we should investigate the subject more in depth in the future. It may be a good path for improving the ability to assess the robustness of a network proven with sound and complete solvers. If we think about it, an

interesting use for such solvers could also go the other way and serve as a reliable tool that could prove the lack of robustness of a neural network. This opens us to an infinite set of possible applications and shows that the field related to binarized neural networks is only at the starting point. It is up to us to explore the many different challenges that this field has in store for us.

Bibliography

[1] Nina Narodytska, Shiva Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. Verifying properties of binarized deep neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[2] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[3] Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. Binary neural networks: A survey. *Pattern Recognition*, 105:107281, 2020.

[4] Kai Jia and Martin Rinard. Efficient exact verification of binarized neural networks. *Advances in neural information processing systems*, 33:1782–1795, 2020.

[5] Guy Katz, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, et al. The marabou framework for verification and analysis of deep neural networks. In *International Conference on Computer Aided Verification*, pages 443–452. Springer, 2019.

[6] Guy Amir, Haoze Wu, Clark Barrett, and Guy Katz. An smt-based approach for verifying binarized neural networks. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 203–222. Springer, 2021.

[7] Webpage of the larq bnn tutorial. `https://docs.larq.dev/larq/tutorials/mnist/`. Accessed: 2022-12-11.

[8] Eyyüb Sari, Mouloud Belbahri, and Vahid Partovi Nia. How does batch normalization help binary training? *arXiv preprint arXiv:1909.09139*, 2019.

[9] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[10] Alexander Borisenko and Kostiantyn Drach. Extreme properties of curves with bounded curvature on a sphere. 2016.

[11] Plumerai website. `https://plumerai.com/#home`.

[12] Wikipedia webpage of smt. `https://en.wikipedia.org/wiki/Satisfiability_modulo_theories#:~:text=SMT%20solvers%20are%20tools%20which,program%20verification%2C%20and%20software%20testing`. Accessed: 2022-12-11.

[13] Clark Barrett. Smt solvers: Theory and practice. `https://resources.mpi-inf.mpg.de/departments/rg1/conferences/vtsa08/slides/barret2_smt.pdf`. Accessed: 2022-11-10.

[14] Marabou github. `https://github.com/NeuralNetworkVerification/Marabou`. Accessed: 2022-09-29.

[15] Marabou python documentation. `https://neuralnetworkverification.github.io/Marabou/`. Accessed: 2022-10-05.

[16] Marco Casadio, Ekaterina Komendantskaya, Matthew L Daggitt, Wen Kokke, Guy Katz, Guy Amir, and Idan Refaeli. Neural network robustness as a verification property: a principled case study. In *International Conference on Computer Aided Verification*, pages 219–231. Springer, 2022.

[17] Serdar Tasiran (Editor) CAV (Conference), Isil Dillig (Editor). *Computer Aided Verification*. Springer, Cham, Switzerland, 2019.

[18] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, pages 39–57. Ieee, 2017.

[19] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[20] Gurobi website. `https://www.gurobi.com/`. Accessed: 2022-11-02.

[21] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[22] Retreieving the binary weights guide. https://docs.larq.dev/larq/guides/bnn-optimization/#retrieving-the-binary-weights.

[23] Webpage explaining how to save a model in protobuf format. https://www.tensorflow.org/tutorials/keras/save_and_load. Accessed: 2023-01-06.

[24] Google colab. https://colab.research.google.com/.

[25] Dnn model for the cifar-10 dataset. https://www.kaggle.com/code/sid2412/cifar10-cnn-model-85-97-accuracy. Accessed: 2022-12-07.

[26] Joseph Bethge, Haojin Yang, Marvin Bornstein, and Christoph Meinel. Back to simplicity: How to train accurate bnns from scratch? *arXiv preprint arXiv:1906.08637*, 2019.

[27] Milad Alizadeh, Javier Fernández-Marqués, Nicholas D Lane, and Yarin Gal. A systematic study of binary neural networks' optimisation. In *International Conference on Learning Representations*, volume 63, page 81, 2019.

[28] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: a calculus for reasoning about deep neural networks. *Formal Methods in System Design*, pages 1–30, 2021.

[29] George Dantzig. Linear programming and extensions. In *Linear programming and extensions*. Princeton university press, 2016.

[30] Zhewei Yao, Amir Gholami, Peng Xu, Kurt Keutzer, and Michael W Mahoney. Trust region based adversarial attack on neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11350–11359, 2019.

[31] Yaguan Qian, Jiamin Wang, Xiang Ling, Zhaoquan Gu, Bin Wang, and Chunming Wu. An adversarial attack via feature contributive regions. 2020.

[32] End to end guide: Converting and benchmarking a model. https://docs.larq.dev/compute-engine/end_to_end/. Accessed: 2022-10-26.

[33] Andy Shih, Adnan Darwiche, and Arthur Choi. Verifying binarized neural networks by angluin-style learning. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 354–370. Springer, 2019.

[34] Lukas Geiger. tinyml talks lukas geiger: Binarized neural networks on microcontrollers.

[35] Webpage of the boost library installation. https://www.boost.org/users/history/version_1_74_0.html. Accessed: 2022-12-11.

[36] Webpage of the cmake library installation. https://cmake.org/download/. Accessed: 2022-12-11.

[37] Lukas Geiger and Plumerai Team. Larq: An open-source library for training binarized neural networks. *Journal of Open Source Software*, 5(45):1746, 2020.

Chapter 9

Hardware and Software specifications

9.1 Hardware Specifications

The hardware specifications of the computer on which every experiment was performed:

MacBook Air (M1, 2020) :

- OS : macOS Monterey Version 12.4
- GPU : Apple M1 8-Core
- CPU : Apple M1 8-Core
- RAM : 8 GB

9.2 Software Specifications

These are the various library versions that I had to install in order to run the various experiments with Larq and Marabou correctly:

- boost 1.74.0 [35]
- certifi 2022.9.24
- cmake 3.22.1 [36]
- cycler 0.11.0

- keras 2.9.0
- kiwisolver 1.4.2
- larq [37] 0.13.0
- matplotlib 3.5.3
- numpy 1.22.3
- pandas 1.5.1
- pbr 5.6.0
- pillow 9.2.0
- protobuf 3.20.1
- pybind11-2.3.0
- pyparsing 3.0.9
- python 3.10.8
- python-dateutil 2.8.2
- pytz 2022.1
- scipy 1.7.3
- six 1.16.0
- tensorflow 2.9.1

Chapter 10

Appendices

10.1 Adaptation of libraries version

- Replace the boost 1.68 version by the 1.74 and modify all the corresponding lines in the "CMakeLists.txt" file.
- Remove the "-Werror" flag from the "CMakeLists.txt" file
- If you want to run the solver with a "warm start" more memory need to be added to the arrays at lines 2378 and 2379 of the file "src/engine/Engine.cpp". In my case, 2 bit were enough.
- If you want to run the solver by providing a network, a dataset (–dataset), an epsilon (-e), a target label (-t), and the index of the point in the test set (-i) you have to specify as well and modify some code in the readTF() function for Marabou to be able to find the .pb file in your directory.
- If you want to run Marabou with the Gurobi solver, don't forget to update the lines 240 and 242 of "CMakeLists.txt" with the current version

10.2 BinaryNet Architecture

Block name	Output size	Layer definition
quant_conv2d_12	30x30x128	Conv 128, 3, strides 1
batch_normalization_9		
quant_conv2d_13	30x30x128	Conv 128, 3, strides 1
max_pooling2d_6	15x15x128	MaxPool, 2x2, strides 2
batch_normalization_10		
quant_conv2d_14	15x15x128	256, 3, strides 1
batch_normalization_11		
quant_conv2d_15	15x15x256	256, 3, strides 1
max_pooling2d_7	7x7x256	MaxPool, 2x2, strides 2
batch_normalization_12		
quant_conv2d_16	7x7x512	512, 3, strides 1
batch_normalization_13		
quant_conv2d_17	7x7x512	512, 3, strides 1
max_pooling2d_8	3x3x512	MaxPool, 2x2, strides 2
batch_normalization_14		
flatten_2		
quant_dense_6	1024	Dense 1024
batch_normalization_15		
quant_dense_7	1024	Dense 1024
batch_normalization_16		
quant_dense_8	10	Dense 10
batch_normalization_17		
activation_2	10	Softmax

Table 10.1: Summary of the BinaryNet architecture

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**