Heriot-Watt University

Masters Thesis

Neural Network Verification with Imandra

Author:
Rémi Desmartin

Supervisor:
Dr. Ekaterina Komendantskaya

A thesis submitted in fulfilment of the requirements for the degree of MSc. Data Science

in the

School of Mathematical and Computer Sciences

August 2021

Declaration of Authorship

I, Rémi Desmartin, declare that this thesis titled, 'Neural Network Verification with Imandra' and the work presented in it is my own. I confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: *Rémi Desmartin*

Date: *Paris, 16 August 2021*

Abstract

Neural networks have known fast development in the past years; this has lead to their deployment in safety-critical applications like autonomous vehicles or aircraft collision detection systems. Recent research has focused on developing tools to guarantee their reliability.

Some verification tools are based on formal methods. Satisfiability Modulo Theory (SMT) solvers are a type of theorem provers that can automatically verify predicates; by expressing neural networks and their properties as predicates, the latter can be proved. Other verification tools rely on abstraction and offer more scalability. Other research focuses on programming languages to verify neural networks, like functional languages or probabilistic programming. New generation reasoning engines, like Imandra, combine many of these characteristics in one tool: integrated functional language, theorem-proving capabilities, and offer new features like input space segmentation and visualisation tools.

The purpose of this thesis is to evaluate to what extent this new generation of integrated tools can be applied to neural network verification. Throughout this thesis, Imandra is used to represent neural network models and to reason about their properties. The thesis presents a library for representing convolutional neural networks (CNNs) in Imandra. Software for importing existing models into Imandra is also developed. A special Imandra module is dedicated to definitions of several known robustness properties from verification literature. In addition to defining and using known verification properties, we use Imandra to suggest a novel approach to the structural analysis of CNNs. We systematically evaluate the proposed Imandra library for usability, scalability, accuracy and applicability. This helps to make conclusions about the advantages and limits of new generation reasoning tools for neural network verification.

Acknowledgements

First and foremost, I would like to thank my supervisor Prof. Ekaterina Komandanskaya sincerely for her precious guidance and for involving me in the Lab for AI Verification.

I would also like to thank Dr Matthew Daggitt; his advice and his expertise on theorem provers were of great help and I am grateful for them.

I want to offer my special thanks to Dr Grant Passmore and the whole Imandra team for their valuable input and time.

Finally, I want to thank Valentine, my parents and my whole family, as well as Elsa, Cabanes, Chiara, Muhammad and Sofia for their enduring support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Neural networks (NN) is an umbrella term for a range of algorithms that take n-dimensional data as input and output corresponding labels. They have proved to perform exceptionally well for problems such as image classification, natural language processing, and more generally with problems that involve processing noisy data. Until the late 1980s, they have been used mostly for research purposes.

With the advent of backpropagation, it was found that NN could generalize well from low-level data; they were then applied to the real-world problem image classification, in particular handwritten digit recognition for banks and postal services (LeCun et al., 1989). This advance led to a considerable gain in attention to NNs.

Since then, research towards more complex architectures, along with the exponential increase of computing power, have led to a steady increase of their performance, and to the use of NNs in more diverse areas of application. NNs are now being deployed in safety-critical applications, like self-driving cars (Bojarski et al., 2016) or aircraft collision-avoidance systems (Julian et al., 2016).

Convolutional neural networks (CNN) are a type of NN that perform on data structured in multidimensional arrays, most commonly used on images. They were already used in the early examples of NN application to image recognition problem previously mentioned (LeCun et al., 1989), but started to attract more widespread attention when AlexNet, a deep CNN, outperformed the previous state of the art with a 20% better accuracy at the 2010 ImageNet Large Scale Visual Recognition Challenge (Krizhevsky et al., 2012). Since then, CNNs have become the new state of the art for image recognition problems.

One essential condition for the large-scale deployment of NN is to guarantee their reliability; NN verification techniques offer to do just that. In particular, they aim to guarantee one or both of:

- domain-specific properties;
- robustness, which in turn guarantees resistance to outside attacks once they are deployed

To explain property-based verification, consider a model responsible for collision detection in an aircraft. Proving that this model advises to turn right when another aircraft is detected by the left radar with a high level of certainty is an example of property-based verification (Katz et al., 2019). Such verification has proved to be a significant challenge due to the black-box nature of NN, their large size and complex structure.

In classification problems, robustness measures the minimum amount of perturbation that needs to be applied to an example for it to be classified incorrectly; the higher the amount of perturbation needed, the more robust the classification model (Bastani et al., 2016). Even for state-of-the-art NN with good performance, minimal perturbations to an example can lead to misclassification (Carlini & Wagner, 2017). In addition, it has been shown that these minimal perturbations can be computed without having access to a model's parameters, making NN vulnerable to outside attacks (M. Wicker & Kwiatkowska, 2018). Such attacks are known as adversarial attacks.

1.2 Motivation

In recent years, research has been concerned with creating tools for NN verification. Even though a number of these tools have been developed, none of them has become widely accepted. This even inspired recent activity in building tools that incorporate many existing NN verifiers such as NeVer2 (Guidotti et al., 2020).

NN verification tools are based on several different techniques, but they work mainly in the following way: they take as input a trained NN, and a series of properties that we wish to verify. The output expresses if the properties are guaranteed or not. Many tools express the result in terms of bounds instead of a binary guarantee.

Existing tools are evaluated on various performance metrics (scalability, accuracy, etc.) but *usability*, i.e. how easy they are to use by non-verification specialists is an important factor as well (Attala et al., 2020). Indeed, adoption by programmers and members of the community can be a significant bottleneck in the deployment of verified software (Fisher et al., 2017).

Three main families of existing tools can be distinguished, based on different theories: SMT solver based, abstraction based, and those leveraging functional programming.

Boolean satisfiability problem (SAT) solvers are a family of software that can determine for a given Boolean expression if there exist assignments that make it evaluate to *true* (see Example 1.1). SMT solvers are similar to SAT solvers, but they can reason about higher-level theories, like linear arithmetic for real numbers (see Example 1.2). Marabou (Katz et al., 2019) is an example of a NN verification tool based on an SMT solver. This family of tools has the benefit of being complete, but they tend to be computationally intensive and thus limited in scalability.

Example 1.1. *The following two Boolean expressions are satisfiability problems that an SAT solver could solve. Possible solutions are given.*

Boolean expression 1: $\neg a \wedge b$
Solution: an interpretation exists (i.e. Boolean values can be assigned to the variables) that make the expression evaluate to true: $a = false, b = true$

Boolean expression 2: $\neg a \wedge a$
Solution: intuitively, no interpretation exists for this formula.

Example 1.2. *The following predicate is a problem that an SMT solver supporting linear arithmetic with integers could solve. A possible solution is given.*

Predicate: $2x * 3y \leq 0$
Solution: one possible interpretation is $x = -1, y = 1$

Another family of tools is based on abstraction: they create approximations of NN that are easier to reason with. As a result, abstraction-based tools tend to scale better. However, they are not complete so they can be less accurate or give indeterminate answers on some problems.

New generation tools have recently been developed to encompass these capabilities into one single tool: theorem proving with real numbers, integrated programming language, higher-order reasoning, visualisation tools. One of these new generation tools is Imandra (Passmore et al., 2020). It embeds an automated reasoning system and a programming language, ImandraML (IML), based on a subset of OCaml. Even though it is not designed specifically for NN verification, its features make it suitable for this task.

A library already exists to formalise NNs in IML (Imandra, 2019), which makes it possible to use Imandra's features to verify NNs' properties. However, that library has two limitations. Firstly, the NN models are implemented as equations and reduce the neural network verification problem to SMT queries. As a result, this library does not leverage the full capabilities and abstractions of a high-order functional language like IML (see Section 2.6.2). Secondly, this library does not define CNNs, which was one of the challenges we took on in this thesis. In reality, these two points are related: it is difficult to reason about the different matrix operations and layers that constitute a CNN without proper abstractions.

1.3 Aim and Objectives

The aim of this project is to provide a first systematic study of Imandra as a tool for verification of NN and make general conclusions about the benefits and limitations of its application in this domain.

To that aim, the objectives of this project were to:

- O1. Represent NN in IML using standard functional language constructs, compare it with a low-level implementation of NN in the existing IML library (see Section 4.2).
- O2. Implement a novel Convolutional NN library in Imandra (see Section 4.2).
- O3. Make verification more accessible by creating a program that translates CNNs from Python to IML (see Section 4.2).
- O4. Produce a library that defines several robustness properties used in the verification literature and summarised in (Casadio et al., 2021), and evaluate it on some practical verification tasks (see Section 5.1)
- O5. Investigate whether novel verification methods, additional to SMT solving, may arise thanks to the higher-level functional approach of our NN IML library (see Section 5.2).
- O6. Make suggestions for future research.

Results and Project Scope. In the end, the project's main original contribution is two-fold. Firstly, it is a more intuitive and abstract Imandra library for Convolutional Neural networks. Prior to this project, Imandra only had a library that presented a low-level equational approach to formalising fully connected NNs. Instead, we implemented more abstract definitions of matrix operations and different kinds of layers used in a CNN, which has proven to be a very challenging and interesting task (see Section 4.2).

This library defines different CNN layers and operations in a general matter, and can be taken by other researchers for any future development.

Secondly, using the power and better clarity that the CNN formalisation gave us, we were able to venture into reasoning about novel verification properties of neural networks. Namely, we designed a series of successful experiments that show that one can reason about the properties of CNN filters and pooling layers to achieve better explainability and in the future perhaps even verifiability of CNN (see Section 5.2). This method only works as a prototype and calls for future research.

Throughout the project, we used an artificial data set representing a simplified emotion recognition problem. It is composed of 144 images of faces divided in two classes, ”Happy” and ”Sad”. The data set is described in full detail in Section 3.2.3. It allowed us to make experiments in a controlled environment, as we knew desirable properties of the data set in advance. It also helped us to mitigate the well-known issues of bad scalability of verification algorithms. Extrapolation of the project’s results to natural data sets is left for future work.

Chapter 2

Background and Literature Review

This Chapter starts with introducing essential mathematical notation and definitions of NNs and CNNs. These will be used in our Imandra formalisation in the following Chapters. The CNN operations are rarely spelt out fully in machine learning literature, but it was essential for this project to clarify all definitions, which now appear in the Background section. We then proceed to introduce Robustness – one of the most common properties used in NN and CNN verification. This sets up the problem space for our verification project. Sections 2.3 and 2.4 survey the literature in formal verification through deterministic and probabilistic approaches. Section 2.5 briefly surveys the work on explainable AI and its relation to verification. Finally, Section 2.6 introduces Imandra – the verification tool of choice for this project.

2.1 Background

2.1.1 Neural networks

Neural networks are a family of algorithms whose structure is inspired by biological neurons in the human brain. They share two structural characteristics with the human brain:

1. The network learns knowledge from data through a learning process.
2. The learned knowledge is stored by the inter-neuron connections' strengths, called synaptic weights. (Haykin, 2009)

The acquisition of knowledge from data through automated tuning of the weights is usually done using the *backpropagation algorithm*. Though verifying the training process itself is the subject of current research (e.g. (Tassarotti et al., 2021)), this project focuses on already trained networks. The training process and the backpropagation algorithm are therefore not covered here but an in-detail explanation can be found in (Haykin, 2009).

2.1.2 Perceptrons

Perceptrons are the simplest form of a neural network; they can classify input linearly separated into categories. They are based on the McCulloch-Pitts model of a neuron, represented in Figure 2.1.

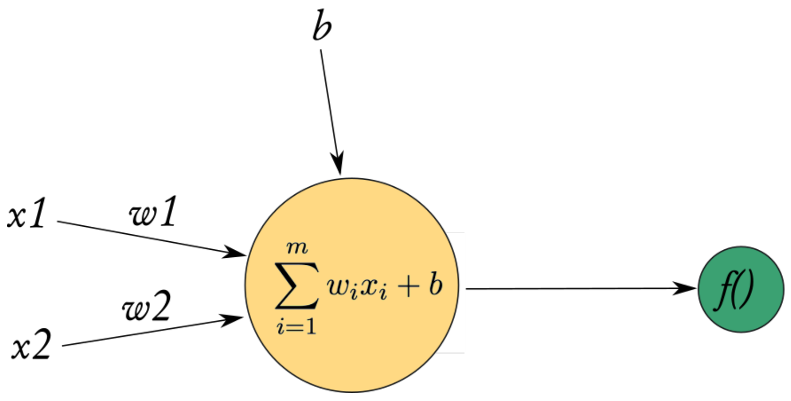


FIGURE 2.1: A graph representation of a perceptron. The inputs (x_1, x_2) are combined with the synaptic weights (w_1, w_2) and the bias b, the activation function f is then applied on the result

In its simplest form, a perceptron behaves like a linear classifier: in order to classify an input vector $(x_0, x_1, ...x_m)$ into one of two classes C_1 and C_2 by computing a linear combination $f(x)$ of the inputs vector with a vector of synaptic weights $(w_0, w_1, ...w_m)$, to which a constant bias b is added:

$$f(x) = \sum_{i=1}^{m} w_i x_i + b \tag{2.1}$$

If the result is positive, it classifies the input into C_1 and if negative into C_2. It effectively divides the input space along a hyperplane defined by:

$$\sum_{i=1}^{m} w_i x_i + b = 0 \tag{2.2}$$

Example 2.1. *Figure 2.2 shows a 2-dimensional input space with 2 classes. In this example, the division hyperplane is the line defined by $y = ax + b$. The perceptron's only weight is a and its bias b.*

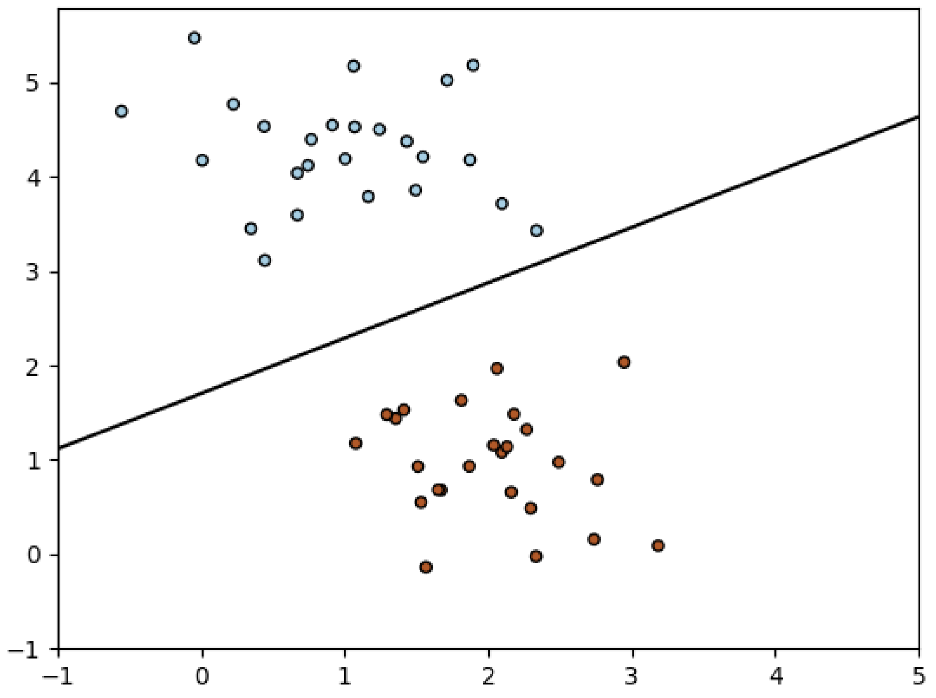


FIGURE 2.2: The division line of a linear classifier.

In most classification problems, classes are not linearly separated. To improve the perceptron's performance for such problems, we can apply a non-linear function a called *activation function* to the linear combination of weights and inputs. The perceptron's output $f(x)$ is then defined as:

$$f(x) = a\left(\sum_{i=1}^{m} w_i x_i + b\right) \tag{2.3}$$

2.1.3 Multi-Layer Perceptrons

Fully connected feedforward NNs, also called multi-layer perceptrons (MLP) are the best-known type of deep NNs. They are made of an input layer, multiple hidden layers and an output layer (see Figure 2.3).

The hidden layers and the output layer are made of nodes, with weighted connections to the previous layers' nodes' outputs. Each node is perceptron described in the previous section. Each layer's output is a vector of its nodes' activations. The weight and biases of all the neurons in a layer can be represented by two matrices noted W and B. By adapting equation 2.3 to this matrix notation, a layer's output L can thus be defined as:

$$L = a(X * W + B) \tag{2.4}$$

Where the operator $*$ denotes that we apply dot-product between X and each row of W, i.e. the sum of the products of their corresponding elements, X is the layer's input and a is the activation function shared by all nodes in a layer.

In an MLP, a layer's input is the output of the previous layer. By denoting a_k, W_k, B_k — the activation function, weights and biases of the k^th layer of a NN respectively, we can define the output of a NN F with L layers for an input X as:

$$F(X) = a_L[B_L + W_L(a_{L-1}(B_{L-1} + W_{L-1}(...(a_1(B_1 + W_1X)))))] \tag{2.5}$$

Hence an MLP can be defined as a function:

$$F(x) : \mathbb{R}^n \to \mathbb{R}^m \tag{2.6}$$

where n is the size of the input and m is the number of neurons in the output layer.

MLP as a classifier. An m-labels classifier $C(x)$ uses a NN with an output layer with m neurons and with the $softmax$ activation function. The output $y = (y_0, y_1, ..., y_{m-1}, y_m)$ is a vector of size m such that $\sum_{i=1}^{m} y_i = 1$ and for $i = 1, ..., m$ we have $0 \leq y_i \leq 1$. The classifier returns only the highest value of the NN's output vector, so that

$$C(x) = argmax F(x) \tag{2.7}$$

Example 2.2. *Figure 2.3 shows a schema of a NN based classifier for the MNIST dataset. The MNIST dataset is composed of 28x28 pixel images of hand-written digits. The pixel values are passed as input to the classifier as a 1-dimensional array of size 784. From this input, the hidden layer's activation and the output are computed, and the output is passed to the softmax layer. The predicted label is the one with the highest score.*

MLPs have several limitations: their large number of parameters makes their training and execution computationally intensive. This, added to their non-linearity, makes them

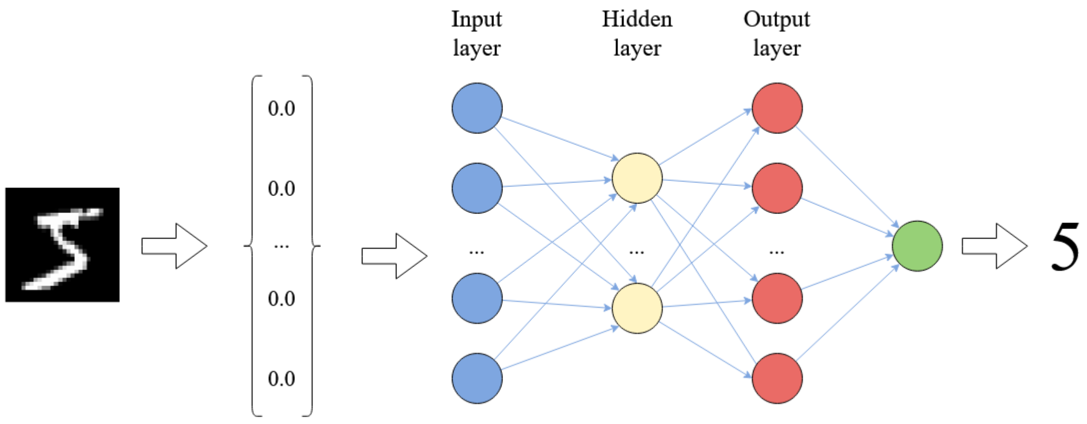


FIGURE 2.3: A NN classifier for the MNIST dataset.

hard to reason about. In addition, as seen in Example 2.2, they flatten images in order to classify them, losing spatial information in the process.

2.1.4 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a type of NNs designed to handle 2-dimensional or 3-dimensional data.

They have layers of different types, namely convolutional, pooling, dropout, and fully connected. We will first describe each of these layer types and then discuss CNNs' global architecture.

2.1.4.1 Convolutional layer

Convolutional layers' weights are represented in the shape of multi-dimensional matrices, called *filters*.

Since a convolutional layer usually has multiple filters, its weights are a set of matrices.

A convolutional layer's output is the result of convolution operations between the input image and each of its filters. That is, for each filter, it outputs one 2-dimensional array called a *feature map*.

A convolution layer is defined by a convolution operation, as follows:

Definition 2.1 (A Convolution Operation)**.** Consider I, a 2-dimensional matrix of size $h_I \times w_I$, and a 2-dimensional square filter F of size f, then M is the result of the convolution operation between I and F, defined as follows:

M's dimensions are $(h_I - 1) \times (w_I - 1)$, and the value of its element at the intersection of the i^{th} row and j^{th} column is determined by the equation:

$$m_{i,j} = \sum_{k=1}^{m} \sum_{l=1}^{n} F_{k,l} I_{(i+k),(j+l)} \tag{2.8}$$

By using $X[i_1, i_2; j_1; j_2]$ to denote the submatrix of a matrix X formed by the intersection of the rows i_1 to i_2 and columns i_2 to i_3, we can rewrite Equation 2.8 as:

$$m_{i,j} = F \cdot I[i, i+f, ; j, j+f] \tag{2.9}$$

Where $\cdot$ is the dot product between two matrices of identical dimensions. M is called a *feature map.*

Example 2.3. *Figure 2.4 shows one step of a convolution operation between an image from our dataset and a filter.*

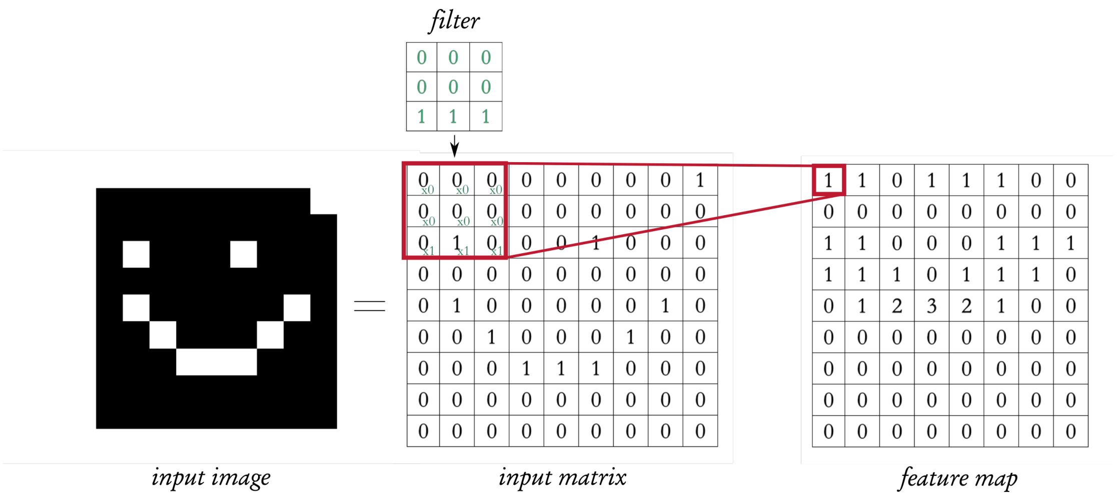


FIGURE 2.4: Example of a convolution operation

The filters (or weights) of convolutional layers are trained using the same backpropagation algorithm as the one used for MLPs, mentioned in the beginning of this Section.

Filters are also called *feature detectors*. This name indicates that they correspond to features, i.e. patterns characteristic of a class's members. Learning features that are independent and discriminating is the goal of the learning process. A convolution operation between filters and an input matrix can be seen as checking which part of the input resembles the filter's feature; hence the name feature map for its result.

Example 2.4. *In figure 2.5, a filter has learned a horizontal line pattern. The blue area of the input image matches the filter better than the red one. As a result, the corresponding value for the blue area in the feature map is higher.*

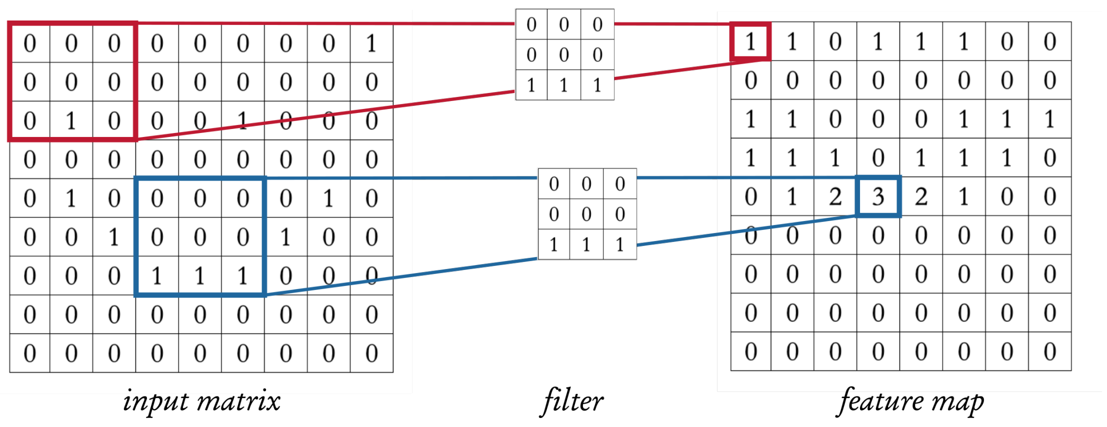


FIGURE 2.5: Feature Map resulting from a convolution operation.

In addition to weights, a bias may be used similarly to fully connected layers. There is one single weight per filter, added to each element of the feature map. Considering a bias b, Equation 2.9 becomes:

$$m_{i,j} = K \cdot I[i, i+f, ; j, j+f] + b \tag{2.10}$$

Finally, an activation function can be used to introduce non-linearity. It is simply applied to all elements in the feature map. Considering an activation function a, we now have:

$$m_{i,j} = a(K \cdot I[i, i+f, ; j, j+f] + b) \tag{2.11}$$

Informally speaking, the thoroughness of the image's coverage by the filters in a convolution operation is inversely proportional to the resulting filter map's dimensions. In order to cover the borders of the image thoroughly and to avoid losing information, the input can be *padded* with rows and columns of null values. The downside is that the feature map for a padded input is larger, which results in more parameters in the network.

On the other hand, to reduce the feature map's size when dealing with large images, the filter can move by a distance of more than one between each application. This distance is called the *stride*. With a bigger stride, since the filter is not applied to all possible regions in the input, some information is lost.

Formally speaking, taking padding of size p and a stride of size s into account, for an input of dimensions $h_I \times w_I$, a filter of size f, a feature map's dimensions are $h_M \times w_M$ where:

$$\begin{aligned} h_M &= \frac{h - f + 2p}{s} + 1 \\ w_M &= \frac{w - f + 2p}{s} + 1 \end{aligned} \tag{2.12}$$

For each element $m_{i,j}$ of M, we have:

$$m_{i,j} = a(K \cdot I[si, si + f; sj, sj + f] + b) \tag{2.13}$$

In the formulas used above, the input matrix only has one channel, but images usually have three and this number increases as the image's representation is passed down the CNN's layers. To apply a convolution operation to input with multiple channels, the filters must have the same number of channels. The resulting feature map only has one channel.

Formally, let us consider an image I with c channels, a filter K of size f with c channels as well. Elements of M are computed by:

$$m_{i,j} = \sum_{l=1}^{c} a(I^l[si, si + n; sj, sj + m] \cdot K^l + b) \tag{2.14}$$

where I^l and K^l denote the input matrix and the filter's k^{th} channel respectively.

Note that in this case, there is still only one bias per filter, which is shared between channels. Since the channels are "flattened" in the operation, the resulting feature map's dimensions are still the same as in Equation 2.12.

We define the result of a convolution operation between an input and a filter as the matrix of dimensions $h_M \times w_M$ as given in Equation 2.12 whose elements are computed by Equation 2.14.

A convolutional layer's parameters are the number of filters n and their size f, its activation function, the padding p and stride size s. For an input matrix of dimensions

(h_I, w_I, c_I), it produces output of dimensions (h_M, w_M, c_M) where

$$\begin{aligned} h_M &= \frac{w_I - f + 2p}{s} + 1 \\ w_M &= \frac{h_I - f + 2p}{s} + 1 \\ c_M &= n \end{aligned} \tag{2.15}$$

Pooling Layer

Pooling layers allow for further downsampling. They usually come after convolutional layers; their input is the feature maps from the previous layer, and their output is a set of 2-dimensional matrices of lower size (see Figure 2.6).

Two main types of pooling operations are used in CNNs: max pooling and average pooling.

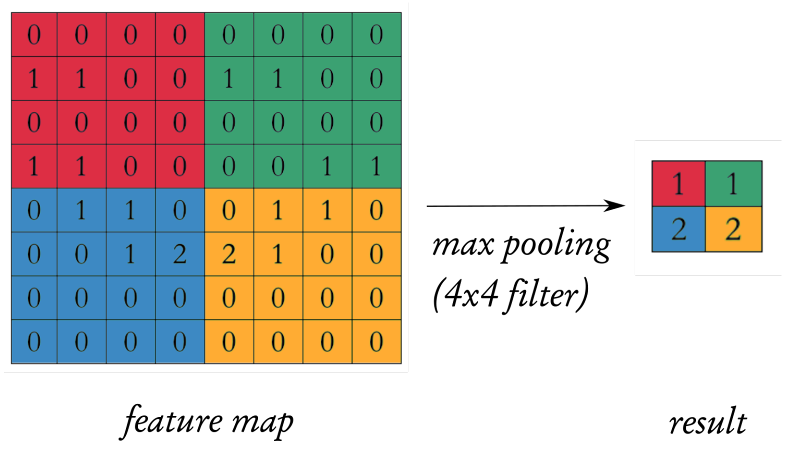


FIGURE 2.6: Max pooling operation with a 4×4 filter and a stride of 4. Each coloured zone is a region where the filter is applied.

Definition 2.2 (Max Pooling Operation)**.** Given a 2-dimensional input matrix I, the *max pooling operation* is a function that returns the output matrix M, such that each element of M is computed the following way: $m_{i,j} = max(I[i, i+f; j, j+f])$ where f is the size of the layer's filter, and *max* is a function that returns the greatest value in a matrix.

Average pooling layers work the same way, but instead of a `max` function, they use an averaging function. Max pooling is used the majority of the time because it reveals the most important features.

A pooling layer does not have trainable parameters. The only hyperparameters it has is the size of its filters, and the stride.

By reducing the size of feature maps, pooling layers reduce the number of parameters in the following layers. By dropping some values, they also help prevent over-fitting.

2.1.4.2 Flatten and Fully Connected Layers

The last layers of a CNN work like an MLP. The flattening layer flattens the 3-dimensional representation into a 1-dimensional array, and several fully connected layers are given by the same operations on matrices as has been defined in Section 2.1.3 where we defined MLPs.

2.1.4.3 CNN Architectures

As mentioned in previous sections, CNNs alternate between convolutional layers and pooling layers. This allows them to detect features on the image and to only keep the parts of the resulting feature maps with relevant information. The typical CNN architecture usually chains several convolutional layers and pooling layers.

In the end, the feature maps volume is flattened and is passed to a MLP. For classification tasks, this MLP's last layer is a $softmax$ layer. This is a standard operation, and was formally defined in the earlier sections.

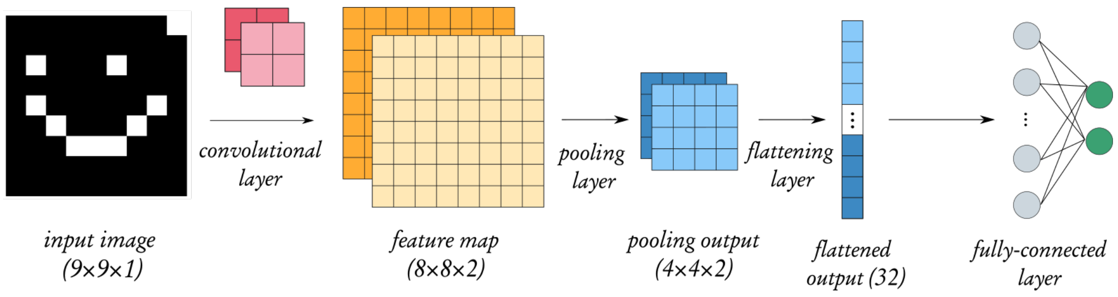


Figure 2.7: Representation of a CNN's layers and intermediate outputs with their dimensions.

Example 2.5. *Figure 2.7 shows the architecture of a simple CNN trained on our simplified emotion recognition dataset, containing one convolutional layer, one pooling layer, and one fully connected layer.*

2.2 Ensuring Robustness of Neural Networks

A NN's robustness is its ability to correctly classify an input to which a perturbation is applied. Verification efforts have focused on this property because it has been proved that even very accurate NNs are vulnerable to adversarial examples.

2.2.1 Adversarial Examples

Adversarial examples for neural networks were first discussed in (Szegedy et al., 2014). This paper shows that it is possible to apply a small perturbation to an image, which leads to an incorrect classification of that image (see Figure 2.8). It was followed by (Goodfellow et al., 2015), who proposes a more efficient way to generate adversarial examples, the fast gradient signed method or FSGM.

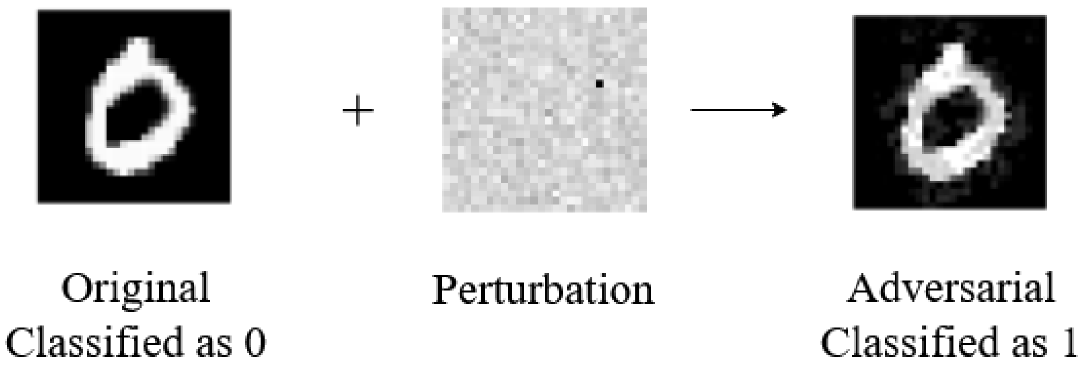


FIGURE 2.8: Application of a small perturbation to an image that results in incorrect classification on a NN trained on the MNIST dataset.

Finding an adversarial example can be expressed as an optimisation problem: considering a classifier function C, an input x and its true class c; for a target class $t \neq c$, find the point x_{adv} closest to x such that $C(x) = t$.

Different definitions can be used to measure the distance between two examples:

- L_0: the number of coordinates that differ between two vectors. In image classification problems, this distance definition measures the number of different pixels between two images.
- L_2: Euclidian distance between two vectors.
- L_∞: the maximum change to any of the coordinates (see Figure 2.9).

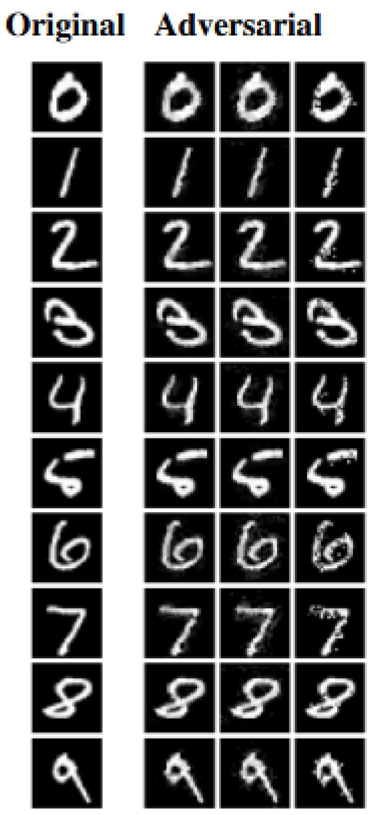


FIGURE 2.9: Adversarial attacks on a NN trained on the MNIST dataset. The first column shows the original image belonging to class l. The three adversarial examples in each row are generated using the L_0, L_2 and L_∞ norm respectively. All three are classified as belonging to class $l + 1$ modulo 10. (Carlini & Wagner, 2017).

As shown in Figure 2.9, adversarial perturbations on images are barely perceptible by humans and are a severe vulnerability for NN-based AI systems, hence the importance of guaranteeing NNs' robustness.

2.2.2 Robustness

Robustness is a property of NNs that can be expressed intuitively as the ability of the network to correctly classify an image that has been perturbed. More specifically, ϵ-ball robustness for an image means that if the distance between the perturbed image and the original is not more than ϵ in the input space, the network classifies the perturbed image correctly.

Example 2.6. *Figure 2.10 shows an image of a hand-written digit "7" from the MNIST dataset. A network robust around this input would classify the perturbed instances within the ϵ-ball as a "7".*

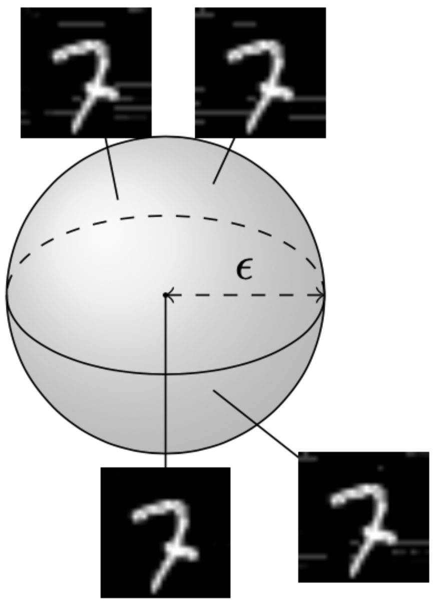


FIGURE 2.10: An image from the MNIST dataset with perturbed images within the ϵ-ball (Casadio et al., 2021)

Different techniques exist to ensure NN robustness during training, like data augmentation (Shorten & Khoshgoftaar, 2019), adversarial training (Madry et al., 2018), or training with logical constraints (Fischer et al., 2019). (Casadio et al., 2021) shows that even though the different techniques generally share the aim of ϵ-ball robustness, this property has not always been formally defined in a unified way and as a result different formal definitions of robustness have been used in the literature.

Classification robustness is the property aimed for by data augmentation. It states that for all inputs x within an ϵ-ball around an input $\hat{x}$, a neural network $f : \mathbb{R}^n \to \mathbb{R}^m$, a classifier C as defined in Equation 2.7 will always classify x as c where c is $\hat{x}$'s class in the input data.

Definition 2.2.1 (Classification robustness)
$CR(\epsilon) \triangleq \forall x : ||x - \hat{x}|| \leq \epsilon \implies argmax f(x) = c$

Standard robustness is the definition implied in adversarial training. This property verifies that the distance between the NN's output for a perturbed instance and the original output y is lower than a value δ; it is defined as:

Definition 2.2.2 (Standard robustness)
$SR(\epsilon, \delta) \triangleq \forall x : ||x - \hat{x}|| \leq \epsilon \implies ||f(x) - y|| \leq \delta$

Lipschitz robustness is inspired by Lipschitz continuity. It states that ”the distance between the original output and the perturbed output is at most a constant, L, times the change in the distance between the inputs”:

Definition 2.2.3 (Lipschitz robustness)
$SR(\epsilon, L) \triangleq \forall x : ||x - \hat{x}|| \leq \epsilon \implies ||f(x) - y|| \leq L||x - \hat{x}||$

Finally, *Approximate classification robustness* is used to train NNs with logical constraints; it only looks at the prediction of the true class c and check whether it is larger than a given value η:

Definition 2.3 (Approximate classification robustness). $ACR(\epsilon, \eta) \triangleq \forall x : ||x - \hat{x}|| \leq \epsilon \implies f(x)_c \geq \eta$

These formalised definitions can be compared in terms of applicable problem domain (general or classification problems only), interpretability, the presence of a loss function or their desirability on all the input space, see (Casadio et al., 2021). In this project they are used to implement robustness checking in IML in Section 4.3.

2.3 Formal Methods for Verification

2.3.1 Formal Methods

Formal methods can be defined as ”mathematical approaches to software and system development which support the rigorous specification, design and verification of computer systems.” (Johnsen, 2021). Theorem provers, software that use induction reasoning to formally prove theorems, are examples of formal method tools.

Formal methods have offered the possibility of software without exploitable bugs for a long time (Rushby & Whitehurst, 1989). However, until recently they had failed to achieve this in real-world scenarios.

(Fisher et al., 2017) gives four reasons why formal verification is relevant today to produce reliable software now more than before. First, formal methods often necessitate searching large spaces and require large memories; the increase of computing power thanks to the fulfilment of Moore’s law in the past decades has made a big difference in this regard.

Second, the increase in formal methods automation allows to prove more complex software properties and faster, especially regarding SAT and SMT solvers (see Examples

1.1, 1.2). Independently of hardware, the performance of SAT and SMT solvers have greatly improved in the past years.

Third, the infrastructure is developed: in the early days, verifying tools had to be verified as they were being developed. Now, formally verified tools like compilers or theorem provers exist. Focus can be put on developing formally verified software.

Last, critical systems that need to be verified are so complex that they necessitate formal verification. The authors give the examples of Amazon Web Services: formal verification methods were used to verify it as such a large system is too big to be verified "manually" (Newcombe et al., 2015).

This last point especially relates to the formal verification of NNs: with thousands or even millions of parameters, most NNs used in real-world applications are too large to be verified manually and call for automated formal verification.

The authors also point out some limitations of formal methods, notably:

L1. Assumptions made by the formal methods must be verified for the result to be valid.

L2. If the model fails to accurately capture the verified program's characteristics, the resulting guarantees may be weaker than needed.

L3. Level of expertise needed and usability: using formal verification tools is usually perceived as hard and necessitating a deep understanding of the field. Tools are often designed for research and not well integrated with other industry-oriented tools.

L4. Performance: formally verified programs are slower.

2.3.2 Formal Verification for Neural Networks

A NN verification tool usually takes a neural network model (for instance represented by its parameters) and a set of properties as input. The verification tools output whether the properties are satisfied for the given network or not (see Figure 2.11).

Several properties have been subject to verification on NNs.

- Robustness, as seen in Section 2.2.2, measures globally how sensitive the network is to input perturbation. It is the most popular.

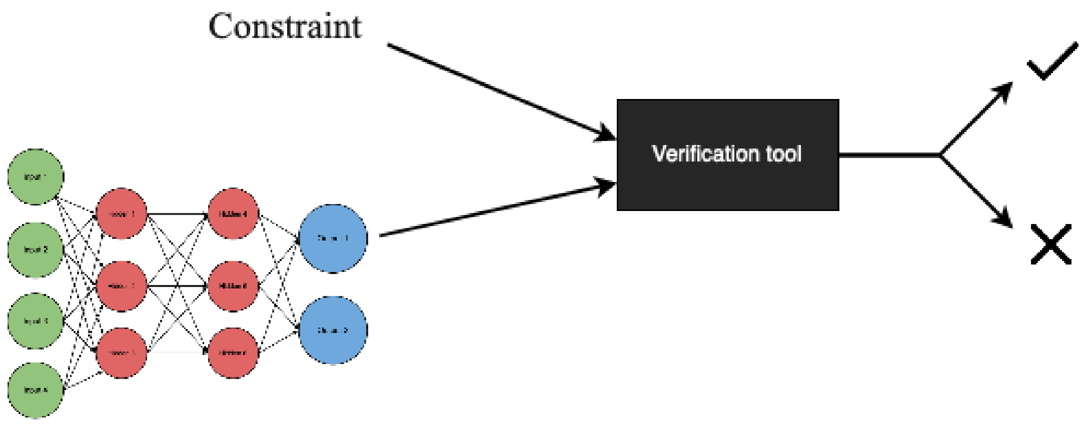


FIGURE 2.11: High-level schema of a NN verification tool.

- Fairness, applied to classification problems, measures if a NN's classification output is affected by different values of a chosen feature. (Urban et al., 2020) proposes a verification tool to certify fairness of feedforward NNs on classification tasks.
- Domain-specific properties have also been verified. The verification benchmark established in (Katz et al., 2017) consists in verifying the behaviour of an aircraft collision detection system depending on specific directional object detectors' data.

This list is not exhaustive as the search for suitable verification properties for NNs is an open research question. This thesis is also making some modest contribution to the search, as will be explained in Section 5.2

SMT-solver Based Tools for Neural Network Verification

Some SMT solver-based verification tools are based on existing general-purpose SMT solvers (Kokke, 2020; Kokke et al., 2020). These tools are limited by the capability of the solver they rely on; for instance, Z3 does not support real numbers.

On the other hand, Marabou (Katz et al., 2019) is based on an SMT-solver designed for network verification. This solver uses the Simplex method and was first proposed in (Katz et al., 2017). Using a specifically designed solver has the benefit of improving performance and scalability, i.e. verifying larger networks. (Katz et al., 2019) also presents a set of benchmark tests consisting of NNs and properties to verify that has been used since then to compare verification tools (Attala et al., 2020; Bunel et al., 2018).

SMT solver based tools are complete, which means that they will never give false-positive results (Bunel et al., 2018). This completeness comes at the price of computation cost:

indeed, SMT solvers are computation-intensive and thus tend to be limited in scalability (Attala et al., 2020). They are also limited to locally linear activation functions (Katz et al., 2017). However, locally-linear properties include ReLU activation function, max pooling and convolution operations, which makes most real-world NNs supported. In addition, techniques to approximate non-linear activation functions with locally linear ones exist which have little effect on the resulting NN's performance (Kokke et al., 2020)

Abstraction-Based Tools for Neural Network Verification

Abstraction-based tools use different methods to approximate NNs and prove properties on those approximations. Reluval (Wang et al., 2018) uses symbolic intervals to approximate NNs. Since the models are simplified and all parameters do not have to be taken into account, it scales quite well. They can also handle non-linear activation functions. However, it is not complete, which means that verifications can be inconclusive (Attala et al., 2020).

ERAN (Singh et al., 2019) works as an output-range analyser: for a given input range, it computes the output's possible range. It supports multiple verification techniques, some of which are complete. It is reliant on an external engine for doing linear programming computations.

Recently, abstraction-based and solver-based verification have been used together to combine the benefits of each one. (Elboher et al., 2020) adds an abstraction-based step to the Marabou toolchain in order to simplify the models that are then verified by an SMT solver-based method.

2.4 Probabilistic Programming for Verification

Probabilistic programming is a programming paradigm in which probabilistic models can be manipulated, like a random variable with a given distribution. Verification tools for probabilistic models exist, such as PRISM (Hinton et al., 2006). It consists of both a language to represent specifications of probabilistic models, such as communication systems with random failures, and a tool to verify these specifications.

Probabilistic programming for verification is the subject of growing interest, as the six papers on this topic that were presented at the POPL21 conference show (POPL, 2021).

2.4.1 Probabilistic Programming for Neural Network Verification

(Cardelli et al., 2019) argues that NNs are probabilistic models, conditioned over the distribution of their training data. They show that probabilistic robustness can be verified for deep NNs.

The fun2model project (Fun2model, 2019), which aims to study probabilistic reasoning for NNs can also be mentioned in this regard. It aims at leveraging probabilistic reasoning to develop novel verifications methods and properties regarding robustness and fairness of complex NN-based decision systems. This shows that new tools open the way for the verification of new properties.

Probabilistic programming languages are linked to functional languages; because of their specific features like being pure, functional languages are usually used for probabilistic programming.

2.4.2 Functional Programming Languages for Verification

Functional programming is another programming paradigm, whose main characteristic is to treat all entities in a program as functions. It relies more on function declarations and executions than on statements. An important characteristic is that they tend to be *pure*: functions have no side effect, they do not modify the values of variables that exist outside of their scope. This makes programs both easy to parallelise and easy to verify.

Many proof assistants and program verifiers are written in functional language because their pure nature makes them less error-prone: Coq (OCaml), F* (F#), Liquid Haskell (Haskell).

2.4.3 Functional Programming Languages for Neural Network Verification

Research is being conducted to embed verification as refinement types in functional programming languages (Kokke et al., 2020). Refinement types are types decorated with boolean-valued predicates that constrain the values described by the type. For instance, they allow building types for ”integers larger than 5” or ”Boolean values that are true”.

The former paper proposes two proof-of-concept libraries that embed robustness requirements as refinement types in Liquid Haskell and F*, two functional languages. This approach greatly benefits usability: the NN and its verification can be expressed in the same programming language. In addition, it offers a developer-friendly interface to SMT solvers as all the verification work is done ”behind the scenes”.

2.4.4 Existing Functional Implementation of CNNs

To the author’s knowledge, only one implementation of CNNs in a functional programming language has been made before (Owl, 2021). This library implements Mask R-CNN, an object detection CNN used in computer vision. It is based on OWL, OCaml’s scientific computing library. It is an efficient and comprehensive implementation of CNNs, but it is unrelated to verification efforts.

2.5 Explainable AI

As discussed earlier, it is hard to understand and reason about NNs’ decision process, which harms the trust in AI systems that rely on NNs. Explainable AI is the idea of making AI systems’ black box more transparent.

Explainability shares verification’s goal to increase trust in NNs. It consists in representing in a human-readable form part of a NN algorithm’s decision process, so that the software’s end-user can understand if the NN’s decision is valid or if it is biased by noise or adversarial perturbations.

Several techniques are used to explain a NN’s choices. (Ribeiro et al., 2016) is one of them; it allows to visualise on which part of an input an image classifier relies the most to make its prediction (see Figure 2.12).

Explainability is complementary to verification, in that explanations can reveal properties about the model’s behaviour, which can then be formally verified (Gopinath et al., 2019), (Kim et al., 2019).

2.6 Imandra

Imandra is a new-generation reasoning software developed by Imandra, Inc. (Passmore et al., 2020). It includes a general-purpose reasoning engine, Imandra Core, and a programming language that allows users to interact with the core. It is an established tool with industrial uses, notably in the financial industry where it is used to verify transaction software and perform validations of trading operations.

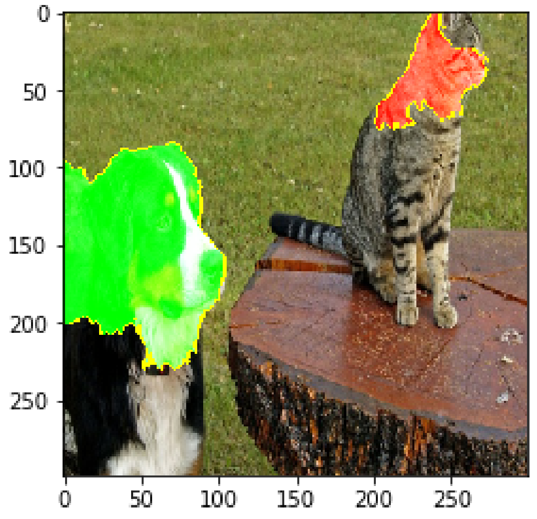


FIGURE 2.12: LIME explanation of a NN's decision to classify an image as "Bernese mountain dog": the regions of the input image that influence positively and negatively the network's classification are highlighted in green and red respectively (Ribeiro et al., 2016)

2.6.1 Imandra Core

Imandra Core is the reasoning engine behind Imandra. It is inspired by Boyer-Moore's work like the theorem prover ACL2 (Imandra, 2021c). The defining feature of this type of theorem provers is the ability to do proofs by induction reasoning. Imandra uses induction at the end of a series of automated proof strategies (see Figure 2.13).

Even though it is not part of its default strategies, Imandra can also call an SMT-solver, which only supports integers. The defining feature of SMT-solvers is their ability to solve large linear arithmetic problems. However, they tend to support a limited number of data types and can solve less general problems.

In formal verification tools, we see a trade-off between the ability to tackle large problems automatically and support for richer data types and proofs by induction. Imandra's balance is close to ACL2 in this regard (see Figure 2.14).

In addition, it supports region decomposition: for a given function that takes a multi-dimensional input (like a classifier), it decomposes the input space into regions from which all input values will lead to the same output (see Figure 2.15).

It can be interacted with like an interactive theorem prover through a command-line interface (CLI). The interface that interests us the most is through a specifically designed functional programming language, IML. Code written with IML can then be exported to regular OCaml.

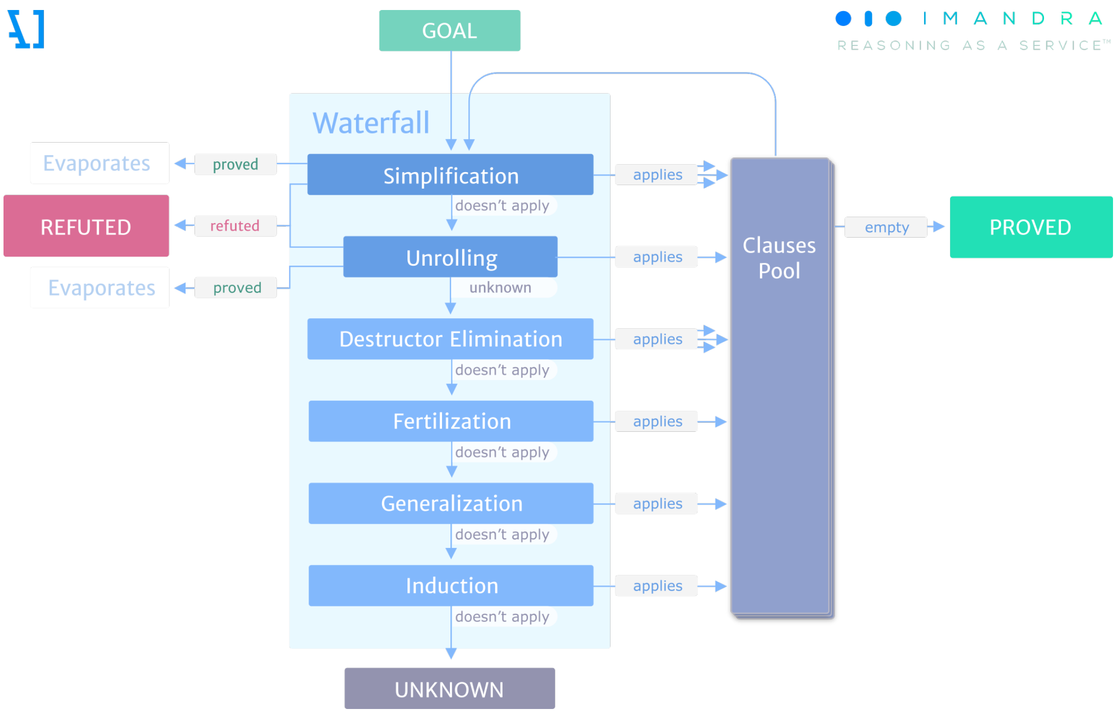


FIGURE 2.13: Schema of the automated use of multiple proving strategies by Imandra Core (Imandra, 2021c).

2.6.2 IML

IML is a programming language integrated to Imandra Core. It is a pure subset of OCaml. Listing 2.1 shows the definition of the ReLU function in IML, which is the same as plain OCaml code. Internally though, IML uses its own formally verified types and operators.

```
let relu x =
  if x <. 0.
  then 0.
  else x
```

LISTING 2.1: Implementation of the ReLU function in IML

Imandra Core API

IML interacts with Imandra Core through a set of functions. These higher-order functions take as input a predicate, i.e. a function that takes at least one parameter and return a Boolean value. The predicate's parameters are the variables that Imandra will reason about. Predicates can also be higher-order and take functions as parameters.

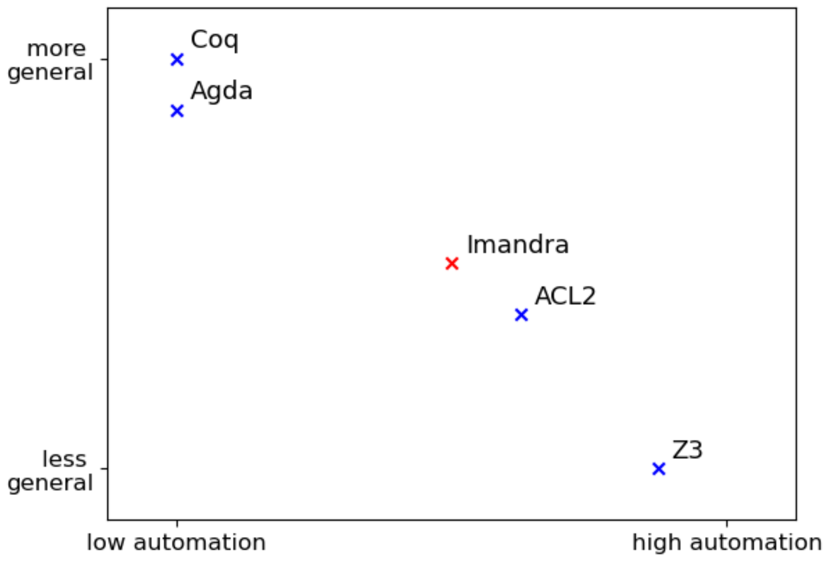


Figure 2.14: Comparison of Imandra with existing verification tools. Note that some tools like Imandra or Agda include SMT solving but it is not a defining feature.

The `instance` function takes a predicate as input and tries to find whether instances that satisfy this predicate exist. Listing 2.2 shows how Imandra can reason about the `relu` function defined in Listing 2.1 thanks to the `instance` function.

```
instance (fun x -> relu x = 100.)
```

Listing 2.2: Use of `instance` in IML

The `verify` function takes a predicate that represents a property as input and tries to verify it. Listing 2.3 shows the verification of a simple property on the `relu` function.

```
verify (fun x -> x <=. relu x)
```

Listing 2.3: Use of `verify` in IML

If the property is not verified, Imandra provides a counterexample. It can be accessed through the `CX` module and it is an executable object within the runtime. Listing 2.4 shows the verification of an incorrect property. By calling the `CX` module, the last command will return the value of a counter example, in this case `-1.0`.

```
verify (fun x -> x >=. relu x)
CX.x
```

Listing 2.4: Use of `verify` and accessing counterexamples in IML

The `theorem` function takes a name, variables, and a function of these variables to prove. If the theorem is proved, it is added to the Imandra runtime and can be re-used in subsequent proofs.

The `lemma` and `axiom` functions work in the same way as the theorem, but the `lemma` is reserved for "smaller" theorems and the `axiom` accepts the property as true instead of trying to prove it.

NN verification

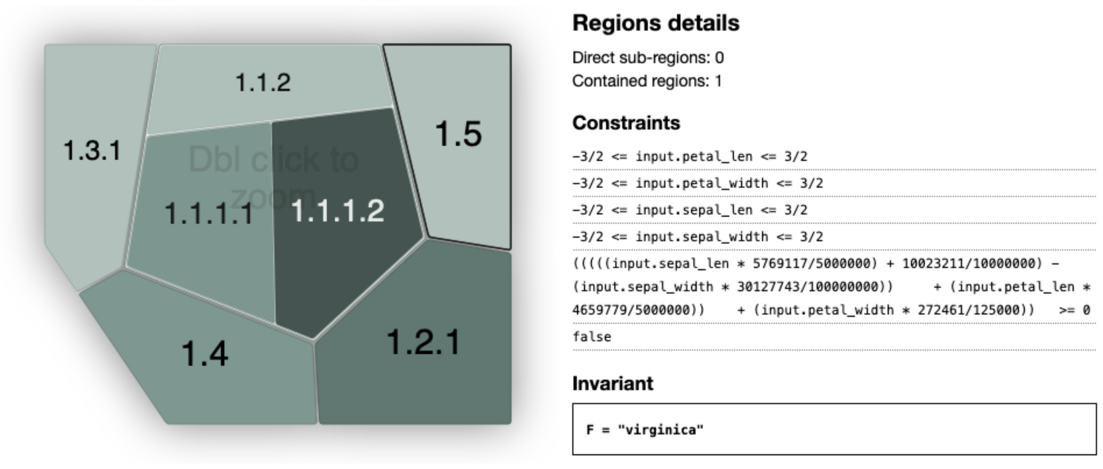
As mentioned before, it is possible to formalise feedforward NNs in IML in order for Imandra to reason about them. The library presented in (Imandra, 2019) takes as input an NNet format file corresponding to a serialised NN and outputs the NN as an IML equation. One can then reason about the resulting equation with Imandra, using the reasoning API described in the previous section. Transforming the NN into an equation leads Imandra to use SMT solving under its Blast strategy.

The representations produced by this library allow Imandra to reason about end-to-end properties of the network, like epsilon-ball robustness. The library has been used to is able to verify properties on the ACAS-Xu models. However, it is limited to fully connected feedforward NN with ReLU activation functions. The NN's weights, inputs and outputs are represented as tuples, which makes it impossible to use IML's standard matrix operations on them. As a result, it is impractical to prove properties on specific parts of a model and to explore alternative verification and explanation methods.

Region Decomposition

IML can use the region decomposition of Imandra Core and visualise the decomposition of high-dimensional input space. Figure 2.15 shows such a visualisation.

This visualisation is not available for higher-order functions as their input space contains functions.



Regions details

Direct sub-regions: 0
Contained regions: 1

Constraints

```
-3/2 <= input.petal_len <= 3/2
-3/2 <= input.petal_width <= 3/2
-3/2 <= input.sepal_len <= 3/2
-3/2 <= input.sepal_width <= 3/2
(((((input.sepal_len * 5769117/5000000) + 10023211/10000000) -
(input.sepal_width * 30127743/100000000))     + (input.petal_len *
4659779/5000000))    + (input.petal_width * 272461/125000))   >= 0
false
```

Invariant

```
F = "virginica"
```

FIGURE 2.15: An example of Imandra's region decomposition on a linear classifier. All inputs within the region defined by the constraints on the right will be classified in the same class ("Invariant").

Probabilistic Reasoning

Even though IML is not a PPL, it is possible to represent random variables and to reason about them. By representing probability mass functions of random variables as high-order functions, it is possible to reason about them.

Example 2.7. *(Imandra, 2021a) shows how Imandra can reason about the famous Monty Hall problem: in a game show, a prize is behind one of three closed doors. The player chooses a door; after he chose, the host – who knows which door hides the prize – opens a "losing" door which the player hasn't picked. The player then has the choice to stick with the first door he chose or to change for the other unopened door.*

By defining functions with verified constraints for user strategies in IML, it is possible to represent them as probability mass functions and to reason about them. Imandra can prove that changing doors leads to a higher probability of winning.

Proving Function Termination

User-defined functions are only accepted by Imandra if it can prove that they terminate. The reason is that it ensures that its logic remains sound, i.e. all theorem provable by this logic must be true (Imandra, 2021b). This aspect of Imandra impacts how functions are implemented in IML (See Section 4.2)

2.7 Conclusion

We have presented the different types of tools that exist for NN verification and analysed the state of the art. None of them manage to completely overcome all the limitations mentioned in Section 2.3.1. The principal existing tools and their main characteristics are evaluated regarding the following evaluation criteria, related to the limitations mentioned above:

- Usability: support for existing formats for representing existing NN and properties (L3).
- Scalability: how well they perform for large networks (L4)
- Precision: if the results are accurate enough to provide usable guarantee of the properties. (L1, L2)
- Applicability: the types of NN that can be verified.

Table 2.1 sums up the main existing tools. Only tools that present features of interest to our project have been selected. We can see that they each have interesting capabilities in different domains. One common point that formal verification tools for NNs have is the focus on research tools, made for verification specialists, rather than usability and adoption.

Tool Name	Complete Verification	Incomplete Verification	Probabilistic Verification	Embedded Prog. Language	Support Non-Linear Functions
Marabou	Yes	No	No	No	No
ERAN	Yes	Yes	No	No	Yes
Prism	No	No	Yes	No	N/A
Starchild	Yes	No	No	Yes	Yes
Imandra (general tool)	Yes	Yes	Yes	Yes	No
Imandra (NN library)	Yes	Yes	No	Yes	No

Table 2.1: Comparison table for the main NN verification tools.

Imandra offers interesting characteristics at the convergence of performance and usability. It offers an embedded programming language with rich data types, and both induction reasoning and SAT-solving. Although probabilistic verification of NNs seems to be an intriguing direction, it is left for future work.

Fully connected NNs can already be formalised in Imandra, as in (Imandra, 2019), but the expressiveness of IML makes it able to both implement MLPs in a more general way, and verify more complex types of NNs like CNNs.

Chapter 3

Methodology

3.1 Overview

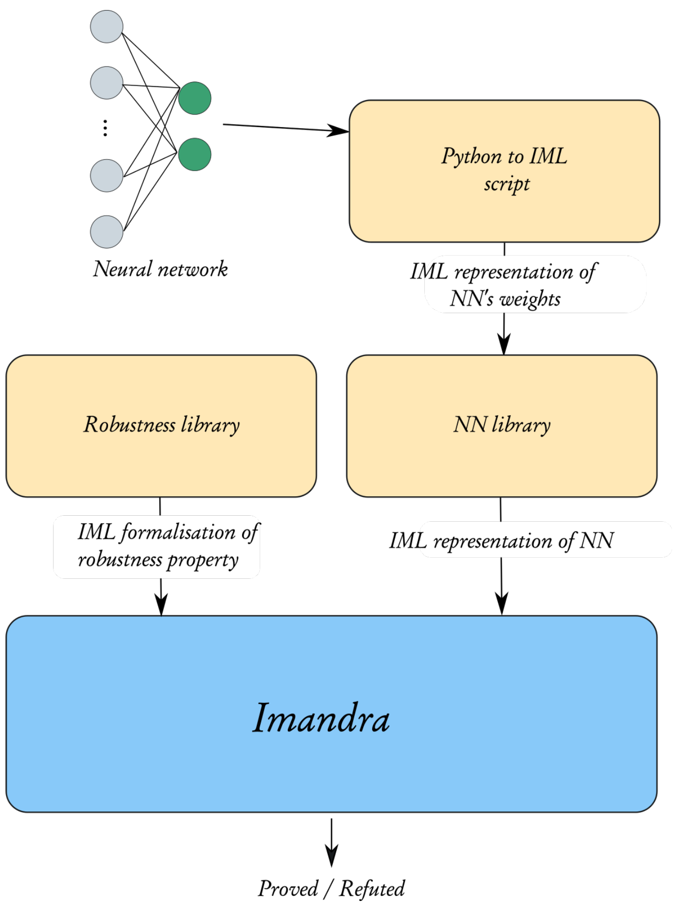


FIGURE 3.1: General architecture of our verification tool.

Our implementation of NN verification in Imandra consists of three main components: the NN library, which allows formalising CNNs, the robustness library, which implements robustness properties that can be verified on NNs and a Python notebook that helps to translate NNs trained with Keras into IML. Both the NN library and the robustness library rely on the same matrix library, which implements matrices and matrices operations (see Figure 3.1).

As this project consists of research as well as programming part, it is important to outline the aims we want to achieve in the form of requirements. The following tables 3.1 and 3.2 present the functional and non-functional requirements for this project. All requirements have been completed.

Requirement	Details	Priority	Completed
NN verification library	Build a library using IML's built-in list type to use a general NN formalisation library in Imandra	Mandatory	Yes
Use Blast	Use Imandra's SMT-solver ability through the Blast strategy	Mandatory	Yes
Support for CNNs	Implement CNN specific operations like convolution and max-pooling to allow our library to formalise them	Optional	Yes
Evaluation	Evaluate our library against criteria we designed	Mandatory	Yes
Import NN into Imandra	Convert existing NN and use our new library to formalise them in IML	Mandatory	Yes
Investigate structural properties of CNNs	Use our library to investigate structural properties of CNNs	Optional	Yes

Table 3.1: Functional requirements.

3.2 Evaluation Criteria and Data Sets

We evaluate our results in three different ways. Firstly, we will use two benchmark data sets and one original data set to run and test the novel Imandra library. Secondly, we will use several known verification properties (such as Robustness, see Section 4.3), as well as define some novel verification properties (Section 5.2). Finally, we co-simulate

Requirement	Details	Priority	Completed
Use IML	The tool should be implemented in the IML programming language	Mandatory	Yes
Evaluation	Evaluation criteria should reflect existing benchmarks	Mandatory	Yes
Documentation	The verification tool's API should be fully documented	Mandatory	Yes

TABLE 3.2: Functional requirements.

our Imandra implementation against standard Python library for CNNs and check that the outputs are identical. This section defines the data sets that we use for evaluation.

3.2.1 ACAS Xu data set

The most common benchmark for NN verification is the one presented in (Katz et al., 2017). Since its first use, it has become a standard benchmark against which to test new verification tools. It consists of a series of 45 MLP with 6 hidden layers of 50 units, designed for the ACAS Xu aircraft collision detection system. It is an interesting case study because it represents a real-world example where formal verification of NNs is important. At the same time, it presents a good balance between accuracy and scalability (Bunel et al., 2018). The NNs also have the advantage of using the ReLU activation function, a locally linear function, which makes them easier to reason about.

The dataset is composed of 7 measures of the relative positions and speeds of the aircraft where the NNs are embarked, called *ownship*, and a potential other aircraft detected by ownship's sensors, called *the intruder*. These measures are:

- ρ: the distance between ownship and the intruder.
- θ, the angle of the intruder relative to ownship's direction.
- ψ, the intruder's direction angle relative to ownship's direction.
- v_{own}, v_{int}, ownship's and the intruder's speed respectively. (see Figure 3.2)
- τ, the time until "loss of vertical separation"
- a_{prev}, the previous advisory given by the system.

It has five different outputs which represent possible advisories for a change in the ship's horizontal direction: Clear-of-Conflict (COC), weak right, strong right, weak left, or

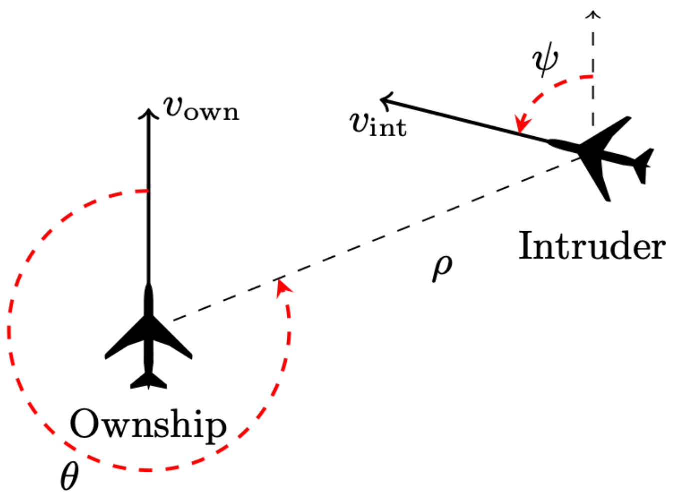


FIGURE 3.2: Representation of ACAS Xu's dataset measures (Katz et al., 2017)
.

strong left. The 45 networks were created by discretising a_{prev} and τ and training a network for each pair of discretised values. The benchmark includes five domain-specific properties. All properties do not apply to all networks.

The issue of linear solvers' scalability for NN verification is well-known; as such, we expect our library to be able to formalise models and properties from the ACAS Xu benchmark, but we do not expect it to be able to verify them. Our focus is on demonstrating IML's expressivity, rather than performance.

3.2.2 Iris Dataset

We used the Iris Dataset, a well-known classification example. in order to conduct initial experiments with Imandra. It is composed of three classes (*setosa*, *versicolor*, *virginica*) with 50 instances each. Each instance represents four measures from an iris flower (sepal length, sepal width, petal length, petal width).

It is widely used because of its extreme simplicity; this is what motivated our using it for preliminary experiments.

3.2.3 Simple Emotion Recognition Dataset

In this project, we used an artificial dataset representing a simplified emotion recognition problem. It consists of 144 unique images of dimension $9 \times 9 \times 1$ (see Figure 3.3). The pixel values are binary.

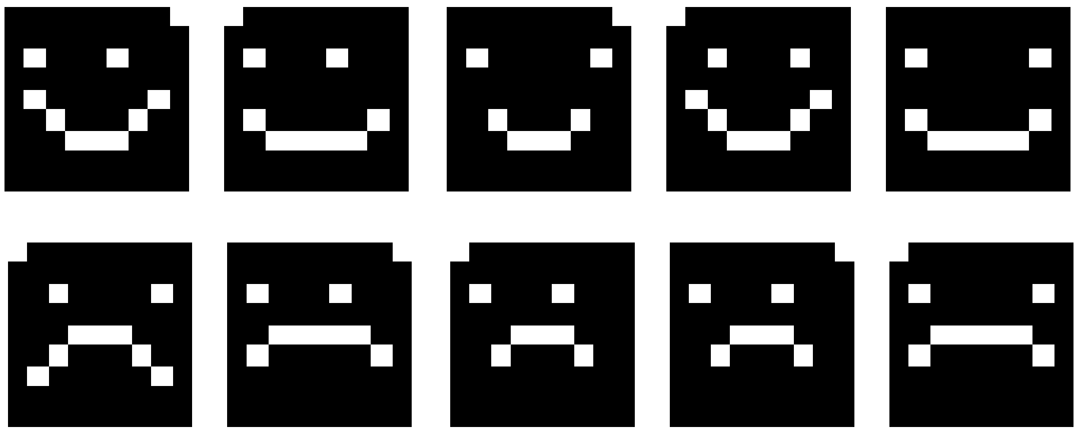


FIGURE 3.3: Sample from the dataset used in the project. The images in the top row are labelled as "Happy" and those in the bottom as "Sad".

As classification problems are often used for neural network verification, and image recognition is the prime problem domain for CNNs, using an image classification dataset made the most sense.

Using artificially small images allowed us to bypass the known performance limitations of verification algorithms. In addition, it allowed us to express domain-specific properties and desirable features for our CNN to learn.

3.3 Software Requirements

The library runs on Imandra. It was developed on a local installation of Imandra v1.0.5.

The Python script to convert Keras models' weights to IML uses Python v3.7.11 with the following dependencies: numpy (1.19.5) tensorflow (2.5.0), scikit-learn (0.22), matplotlib (3.2.2).

It is given in the form of Jupyter notebooks designed for Google Colaboratory.

Chapter 4

Implementation

In this chapter, we try to accomplish Objective 1, the implementation of NNs and CNNs in Imandra. Our implementation represents MLPs in a more intuitive and flexible way than the previously existing library (Imandra, 2019), and that supports CNNs in particular. We pay attention to describing its main features and the reasons behind various programming choices.

All the code presented in this report is publicly available at https://gitlab-student.macs.hw.ac.uk/rhd2000/dissertation.

4.1 Naive Fully Connected Network Implementation

As an initial naive experiment in NN verification, we started this project by defining in IML a simple feed-forward neural network represented in Figure 4.1 on the Iris dataset (see Section 3.2).

This first formalisation consisted in implementing low-level MLPs manually. The result is a similar representation to that of the existing NN library (Imandra, 2019). In this low-level implementation, inputs are represented as the domain-specific data type `iris_input` and weights as sets of real numbers.

```
type iris_input = {
  sepal_len: real;
  sepal_width: real;
  petal_len: real;
  petal_width: real;
}
```

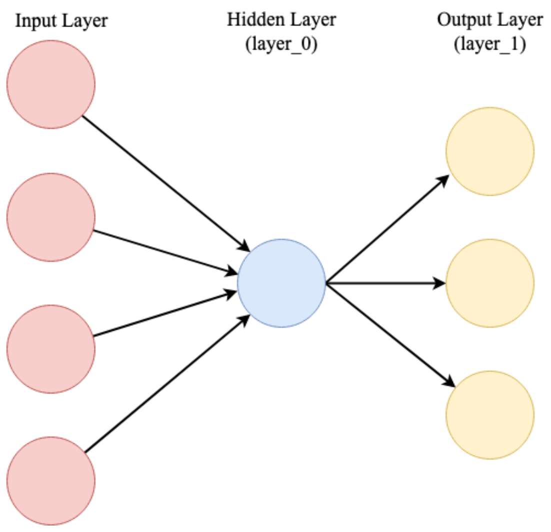


FIGURE 4.1: A graph representation of the simple MLP used in preliminary experiments.

```
let process_iris_input (x: iris_input) =
  let f0 = x.sepal_len in
  let f1 = x.sepal_width in
  let f2 = x.petal_len in
  let f3 = x.petal_width in
  (f0, f1, f2, f3)

let layer_0 (w0, w1, w2, w3, w4) (f1, f2, f3, f4) =
  relu (w0 +. w1 *. f1 +. w2 *.f2 +. w3 *. f3 +. w4 *. f4)

let layer_1 (w0, w1, w2, w3, w4, w5) f1 =
  let o1 = w0 *. f1 +. w1 in
  let o2 = w2 *. f1 +. w3 in
  let o3 = w4 *. f1 +. w5 in
  (o1, o2, o3)

let process_iris_output (c0, c1, c2) =
  if (c0 >=. c1) && (c0 >=. c2) then "setosa"
  else if (c1 >=. c0) && (c1 >=. c2) then "versicolor"
  else "virginica"

let weights_0 = (1.0023211, 1.1538234, -0.30127743, 0.9319558, 2.179688)

let weights_1 = (-2.651993, 0.81521773, -0.83343804,  0.27192873,
    -0.27463955,  -1.21521)

let model input = process_iris_input input |> layer_0 weights_0 |>
    layer_1 weights_1 |> process_iris_output
```

LISTING 4.1: Naive MLP implementation

Note that in Listings, some error handling is omitted for better clarity. For the full implementations, please see the repository.

Imandra is able to reason about an ϵ-ball classification robustness property as expressed in Listing 4.2; it proved that the network was robust around the input defined above for $\epsilon = 0.2$.

```
(* A versicolor instance, normalized values *)
let versicolor_input = {
  sepal_len = 0.31099753;
  sepal_width = -0.59237301;
  petal_len = 0.53540856;
  petal_width = 0.00087755;
};;

let is_valid input =
  -3. <=. input.sepal_len && input.sepal_len <=. 3. &&
  -3. <=. input.sepal_width && input.sepal_width <=. 3. &&
  -3. <=. input.petal_len && input.petal_len <=. 3. &&
  -3. <=. input.petal_width && input.petal_width <=. 3.

let distance x y =
  abs (x.sepal_len -. y.sepal_len) +. abs (x.sepal_width -. y.sepal_width
   ) +.
  abs (x.petal_len -. y.petal_len) +. abs (x.petal_width -. y.petal_width
   )

let epsilon = 0.2

verify (fun x -> is_valid x
  && distance x versicolor_input <=. epsilon
  ==> model x = "versicolor")
```

LISTING 4.2: Naive ϵ-ball robustness verification of a MLP

Note that we verify the robustness for values that are within the observed range in the dataset (checked in `is_valid`) in order to limit the search space. Euclidian distance is used to measure the distance between two inputs.

Since this implementation's approach consists in manually converting trained networks into IML equations, which is a long and error-prone process, it is not scalable. On the contrary, the existing solution, which uses the same approach but generates IML equations automatically, can easily represent larger networks.

However, this general approach lacks a representation of NNs' weights as matrices. It is difficult to represent CNNs, which have a more complex structure than MLPs, and to formalise high-level verification properties without such a representation.

4.2 Convolutional Neural Network Library

In this section, we discuss our CNN library's implementation details and the design choices made during its conception.

CNNs are designed to operate on images, whose representation remain in the form of multi-dimensional arrays throughout the network until the last few layers. We chose to represent these arrays as nested lists, instead of following the existing library's strategy of unfolding computations into linear equations with an external tool. This allows us to express 2-dimensional properties on all parts of the network (filters and intermediate outputs like feature maps).

The `Vector` and `Matrix` modules implement the matrix manipulation operations at the core of our library. Using these, all the necessary types of layers (convolutional, max pooling and fully connected) are implemented in their respective modules. The `Layers` module abstracts these implementations and presents an API of higher-order functions to construct *layer functions* with set weights and parameters. These layer functions can be easily chained thanks to the `Result` module to form a CNN model.

4.2.1 Vectors and Matrices

Vectors represent 1-dimensional arrays and are defined as variable-length non-mutable lists using the built-in type `List`.

```
type 'a vector = 'a list
```

Most list manipulation functions of the OCaml `List` module are available in Imandra, these functions are used whenever possible. Functions were defined in the most general way for more but some, like `sum`, are dependent on the type of data that the list contains.

Matrices represent 2-dimensional arrays and are defined as:

```
type 'a matrix = 'a vector list
```

Each nested Vector represents a row; all rows in a matrix should be of the same length. This can be checked at runtime with the function `is_valid`:

```
let rec is_valid' (m:'a matrix) l = match m with
  | [] -> true
  | (hd::tl) -> List.length hd = l && is_valid' tl l

let is_valid (m:'a matrix) = match m with
  | [[]] -> true
  | (hd::[]) -> true
  | (hd::_) -> is_valid' m (List.length hd)
  | _ -> false
```

Note that functions decorated with a prime (') are usually recursive functions called only by a parent function with the same name.

Dimensions and coordinates in matrices are always given in the form of tuples of two values of the form: `(rows, columns)`. The origin of each matrix's coordinates is in the top left corner.

As there is no built-in type available for matrices equivalent to `List` for vectors, the `Matrix` module implements a number of functions for basic operations needed throughout the implementation. For instance, `map2` takes as inputs a function f and two matrices A and B of the same dimensions and outputs a new matrix C where each element $c_{i,j}$ is the result of $f(a_{i,j}, b_{i,j})$:

```
let rec map2 (f: 'a -> 'b -> 'c) (x: 'a matrix) (y: 'b matrix) = match x
    with
  | [] -> (match y with
    | [] -> Ok []
    | y::ys  -> Error "map2: invalid list length.")
  | x::xs -> match y with
    | [] -> Error "map2: invalid list length."
    | y::ys -> let hd = Vec.map2 f x y in
    let tl = map2 f xs ys in
    Res.lift2 List.cons hd tl
```

This implementation allows us to define other useful functions in a concise way. For instance, the dot-product of two matrices, or the L0 distance between two matrices are defined as:

```
let dot_product (a:real matrix) (b:real matrix) =
  let c = map2 ( *. ) a b in
  Res.map sum c

let l0 m1 m2 =
  let diff = fun a b -> if a = b then 0. else 1. in
  let m3 = map2 diff m1 m2 in
```

```
Res.map sum m3
```

`column` and `nth` help to manipulate functions within matrices by returning a single column of values or a single value with given coordinates.

The Result Type

The library uses the built-in OCaml result type to handle errors. Almost all functions in the library return a type `('a, string) result`. This allows using the built-in functions for handling results. For instance, the bind operator is used to chain multiple functions that return a result, and the final result will be an error message if any one of them fails:

```
open Result
let model input = layer_0 input >>= layer_1 >>= layer_2 >>= layer_3
```

The `Res` module extends the built-in Results module by adding useful functions commonly used with monads, like `bind2`, `lift`, `flatten` or `extract_list`, which transforms a `List` of `Results` into a `Result` of `List`.

Finally, it implements boolean operators (`lt, lte, is_approx, or, and`) in order to compare values of type `(real, 'a) result` and `(boolean, 'a) result`. They are important for expressing verification properties with CNNs' outputs.

4.2.2 Convolutional Layer

The `Convolution` module implements convolutional layers as described in 2.1.4.1. The functions `convolution_row, convolution` and `conv_channels` implement the convolution operation formalised in this section from an input's row to a 3-dimensional input with channels.

`convolution_row` takes as input an input matrix, a filter matrix and a row index, and outputs the result of applying the filter on the input's row of the given index. The loop to apply the filter repeatedly to the input at different coordinates is made by the recursive function `convolution_row'`. Note that the filter is applied from right to left. This is due to the fact that Imandra must be able to prove a function's termination in order to accept its definition. In Listing 4.3, the first definition would not be accepted by Imandra, because in the stopping condition, Imandra does not know the value of `max_column_index`. It is able to prove the termination of the second definition because it compares `column_index` with 0. This pattern is used in several parts of the code for the same reason.

```
(* Imandra cannot prove termination and reject this implementation *)
let rec loop iterator max =
  if iterator > max then []
  else iterator :: (loop (iterator + 1) max)

let integer_list max =
  loop 0 max

(* Imandra accepts this implementation *)
let rec loop' iterator =
  if iterator < 0 then []
  else iterator :: (loop' (iterator - 1))

let integer_list' max =
  List.rev (loop' max)

let rec convolution_row' input filter (row, col) =
  let (row', col') = Matrix.dimensions filter in
  if col < 0 then Ok [] else    (* Imandra can prove that this stopping
    condition is eventually reached *)
  let sub_m = Matrix.sub_matrix input (row, col) (row', col') in
  let dot_p = Res.bind sub_m (fun x -> Matrix.dot_product x filter) in
  let head = convolution_row' input filter (row, col - 1) in (* note that
     the column index decreases *)
  Res.bind2 head dot_p (fun x y -> Ok (x @ [y])) (* note that the list is
     constructed from the right *)

let convolution_row input filter row =
  let (i_rows, i_cols) = Matrix.dimensions input in
  let (f_rows, f_cols) = Matrix.dimensions filter in
  (* Compute the index of the right-most application of the filter *)
  let col = (i_cols - f_cols) in
  convolution_row' input filter (row, col)
```

Listing 4.3: Definitions of `convolution_row`

`convolution` implements a convolution operation between a single-channel matrix and a filter. In its definition, the built-in function `List.fold_left` is used to repeat an action in a loop (see Listing 4.4). This lets us avoid a construct like in `convolution_row` described above, and since `fold_left` is a built-in function its termination is guaranteed. In the accumulation function `acc_fun`, the input is a tuple containing the actual terminator `xs` and an iterator index `i` which is simply incremented.

```
let convolution (input: real Matrix.matrix) (filter: real Matrix.matrix)
    =
  let (i_rows, _) = Matrix.dimensions input in
  let (f_rows, _) = Matrix.dimensions filter in
  let acc_fun (i, xs) =
    if f_rows + i > i_rows
    then (i + 1, xs)
    else  (* skip last rows *)
      let x = convolution_row input filter i in
    (i + 1, Res.bind2 x xs (fun x xs -> Ok (xs@[x]))) in
  let (_, res) = List.fold_left acc_fun (0, Ok []) input in
  res
```

Listing 4.4: Definition of `convolution`

The function `conv_channels` applies convolution to each of the input's channels with the corresponding filter's channel, then adds the resulting matrices together to arrive at a 2-dimensional feature map as described in Equation 2.14.

Note that no weights or activation functions are used, the stride is always 1 and the padding 0.

4.2.3 Max Pooling Layer

The `Max_pool` module implements max pooling layers. Their implementation follows a similar structure as the convolutional layer's. `max_pool_row` applies max pooling to a single row of the input, `max_pool` applies this to all rows.

As for convolutional layers, the max pooling operations are done from the bottom and from the right, decreasing the column and row indices each iteration in order to let Imandra prove termination.

In `max_pool_row` and `max_pool_loop`, it is necessary to check the sign of `pool_cols` and `pool_rows` and return if they are negative because Imandra does not know their sign, so cannot prove the function's termination without these conditions.

```
 let rec max_pool_row input pool_size (row, col) =
  let (_, pool_cols) = pool_size in
  if col < 0 || pool_cols <= 0 then Ok [] else (* pool_cols <= 0
    condition needed for Imandra to prove termination  *)
  let m = Matrix.sub_matrix input (row, col) pool_size in
  let head = Res.map Matrix.max m in
  let tail = max_pool_row input pool_size (row, col - pool_cols) in
  Res.bind2 head tail (fun hd tl -> Ok (tl @ [hd]))
;;

let rec max_pool_loop input (pool_rows, pool_cols) row =
  let (i_rows, i_cols) = Matrix.dimensions input in
  if row < 0 || pool_rows <= 0 then Ok [] else (* pool_rows <= 0
    condition needed for Imandra to prove termination  *)
  let pool_nb = i_cols / pool_cols in
  let col = pool_cols * (pool_nb - 1) in
  let head = max_pool_row input (pool_rows, pool_cols) (row, col) in
  let tail = max_pool_loop input (pool_rows, pool_cols) (row - pool_rows)
     in
  (Res.lift2 List.cons) head tail
;;

let max_pool' input (pool_rows, pool_cols) =
  let (i_rows, _) = Matrix.dimensions input in
  if i_rows < pool_rows then Ok [] else
  let pool_nb = i_rows / pool_rows in
  let row = pool_rows * (pool_nb - 1) in
  max_pool_loop input (pool_rows, pool_cols) row
;;

let max_pool pool_size input = Res.map List.rev (max_pool' input
    pool_size)
```

Listing 4.5: Max pooling implementation

Our implementation uses the width of the filter as its stride value. This allows for maximum size reduction, while not losing any information.

4.2.4 Fully Connected Layer

Module `FC` implements fully connected layers. `fc` takes as parameters an activation function, a 2-dimensional matrix representing the layer's weights and a vector representing the input. Note that each row of the weights matrix represents the weights for one of the layer's node. The bias for each node is the first value of the weights vector, and 1 is prepended to the input vector when computing the linear combination of weights and input to account for that.

```
let activation f w i = (* activation function, weights, input *)
  let linear_combination m1 m2 = if (List.length m1) <> (List.length m2)
    then Error "invalid dimensions"
    else Res.map Vec.sum (Vec.map2 ( *. ) m1 m2) in
  let i' = 1.::i in (* prepend 1. for bias *)
  let z = linear_combination w i' in
  Res.map f z

  let rec fc f (weights:real Matrix.matrix) (input:real Vec.vector) =
    match weights with
  | [] -> Ok []
  | w::ws -> Res.lift2 List.cons (activation f w input) (fc f ws input)
```

Listing 4.6: Fully connected layer implementation

4.2.5 Assembling Layers Into a CNN

The `Layers` module encapsulates the previous modules to expose higher-order functions that instantiate layer functions. Notably, the convolution and max pooling layers are implemented in their respective modules for a single filter. The `max_pool` and `convolution` functions map these implementations over lists of filters and then convert the result from a list of `result` to a `result` of list using `Res.extract_list`.

```
let max_pool pool_size (input: real Matrix.matrix list) =
  Res.extract_list (List.map (Max_pool.max_pool pool_size) input)

let convolution (filters: real Matrix.matrix list list) (input: real
    Matrix.matrix list) =
  Res.extract_list (List.map (Convolution.conv_channels input) filters)
```

Listing 4.7: Definition of the max pooling and convolutional layer constructors

These functions are supposed to be partially applied to a multi-dimensional array of weights, an activation function or hyperparameters to create layer functions that can be chained to form a network (see Listing 4.8).

```
let layer_0 = Layer.convolution Layer0.filters
let layer_1 = Layer.max_pool (2, 2)
let layer_2 = Layer.flatten
let layer_3 = Layer.fc (fun x -> x) Layer3.weights

let arg_max (output: real list) = match output with
  | fst::(snd::[]) -> if fst >. snd then Ok "Happy" else Ok "Sad"
  | _ -> Error "error"

let model input = layer_0 input >>= layer_1 >>= layer_2 >>= layer_3
```

Listing 4.8: Definition of a CNN model using layer functions

As these layer functions are implemented in a general way, an arbitrary number of layers of any type can be chained together as long as the dimensions of each layer's output match those of the next's expected input. For instance, an MLP can be created by chaining only fully connected layers.

We implement an argmax layer specific to our problem domain to obtain a class name as output.

Note that the dimensions of layers' inputs and outputs are not specified, they are deduced from the layer's parameters' dimensions.

4.2.6 Importing Trained Networks

Our library assumes that networks are already trained. For this project, we used Keras to train our networks, and we provide a Python script to convert a NN saved in Keras' format into an IML file containing each layer's weights. The layers must then be instantiated with layer functions (see Listing 4.8).

In order to be compatible with our library, CNNs must take the previously mentioned restrictions into account: they must use no bias and no activation function in convolutional layers, no padding and default stride for convolutional and pooling layers.

4.2.7 Quantised Version

In order to verify robustness, we used Imandra's Blast proof strategy which forces the use of a built-in SMT-solver (see Section 5.1.1). As the Blast strategy only supports integers, we created a copy of our library that uses integers instead of real numbers. The main major change consisted in changing operators and implementing a Euclidian division since the built-in division operator is not supported by Blast as it raises exceptions.

When using this version of the library, the inputs of the NNs' weights must be quantised. For our experiments, we quantised the weights by simply rounding the real-valued weights that resulted from training.

4.3 Formalising Different Robustness Properties

IML's expressiveness allows us to express easily the different definitions of robustness given in (Casadio et al., 2021). Note how close Imandra syntax stands to common mathematical notation in abstract definitions of these properties in Section 2.2.2.

```
  module Robustness = struct
  let l0 x y = (match x with
  | x::[] -> (match y with
  | y::[] -> Matrix.l0 x y
  | _ -> Error "invalid matrix size y")
  | _ -> Error "invalid matrix size x")

  let (>>=) = Res.bind

  let (<=?) = Res.lte

  let (<=??) x y = (match y with
  | Ok y' -> x <=? y'
  | Error _ -> false)

  (* Classification robustness *)
  let cr model input c epsilon ?(constrnt=true) x = constrnt && (l0 x
    input) <=? epsilon ==> model x = (Ok c)

  (* Standard robustness *)
  let sr model input delta epsilon ?(constrnt=true) x =
    let y = model input in
    let fx = model x in
    let dist = Res.bind2 y fx Vec.l0 in
    constrnt && (l0 x input) <=? epsilon ==> dist <=? delta

  (* Lipschitz robustness *)
  let lr model input l epsilon ?(constrnt=true) x =
    let y = model input in
    let fx = model x in
    let dist_out = Res.bind2 y fx Vec.l0 in
    let dist_in = l0 x input in
    let l' = dist_in >>= (fun x -> Ok (x * l)) in
    constrnt && dist_in <=? epsilon ==> dist_out <=?? l'

  (* approximate classification robustness *)
  let acr model input class_id eta ?(constrnt=true) x =
    let class_res = Res.flatten (Res.map (Vec.nth class_id) (model x)) in
    constrnt && class_res <=? eta
end
```

Listing 4.9: "Implementation of the different definitions of robustness"

These implementations make it easy to verify different type of ϵ-ball robustness for a given `model` around an `input`. Notice that the functions can take a `constrnt`, a boolean constraint that inputs must satisfy, in order to reduce the search space.

4.4 Conclusion

Our library implements all the layers necessary to represent MLPs and CNNs in Imandra. In the current state, convolutional and pooling layers lack flexibility as zero-padding, variable stride, the use of biases and activation function is not implemented due to lack of time. As the focus of this verification project is to investigate the possibilities offered by CNN formalisation in IML rather than an exhaustive implementation, we leave these improvements to future work.

Imandra's requirement to prove function termination has proved to be a challenge in the implementation and has guided some of our design choices. The resulting library is accepted by Imandra's logic-mode engine.

IML's expressivity also allows us to implement different definitions of ϵ-ball robustness. Together with the CNN library, we can verify if these properties hold for trained NNs and given inputs.

Chapter 5

Evaluation and Verification

In this chapter, we fulfil Objectives 2 and 3: evaluate how Imandra's capabilities can be used for CNN verification, and explore a novel verification property, specific to CNNs. We first discuss the successful verification of multiple robustness properties on a small network; we then present the exploration of a novel verification property. Finally, we discuss our attempt at verifying properties on a larger benchmark.

5.1 CNN Robustness Verification

5.1.1 Choice of Proof Strategy

With the library described above, we implemented the same simple MLP as the one described in Section 4.1. When we tried to verify the same property, Imandra did not yield the expected result, but instead gave a weaker bounded result:

```
Unknown (verified up to depth 100),
```

This means that Imandra tried to *unroll* a recursive function to find a counter-example, but did not find one after unrolling up to 100 recursions. The unrolling phase comes early in Imandra's default strategy (see Figure 2.13). An unknown result after the unrolling phase means that Imandra did not get past that phase.

Since we knew the problem was tractable by Imandra from our preliminary experiment, Imandra's inconclusiveness came from the way it tried to prove the robustness property when dealing with our library's data types.

In order to force Imandra to use its built-in SMT solver, we use the `[@@blast]` directive to indicate that the Blast strategy should be used. Blast only supports integers, so in order to use it our library had to be modified to use integers. We quantised the trained model's weights by rounding them to fit this integer version of our library. By running the quantised model on our testing set, we verified empirically that quantisation by rounding does not result in a too great diminution of accuracy.

By using the Blast strategy on the quantised model, we can now prove its property and obtain the same verification result as for the low-level implementation.

5.1.2 Description

We trained a CNN with the architecture shown in Table 5.1 on our artificial emotion recognition dataset described in Section 3.2.3. For training, we used the Tensorflow library, through its Keras high-level API in Python. Figure 2.7 is a visual representation of this architecture.

#	Layer Type	Output Dimensions	Parameters #
N/A	Input	$9 \times 9 \times 1$	N/A
0	Convolution	$8 \times 8 \times 2$	8
1	Max Pooling	$4 \times 4 \times 2$	N/A
2	Flatten	32	N/A
3	Dropout	32	N/A
4	Fully Connected	2	66

TABLE 5.1: Test CNN summary

The dataset was split into a training set of 108 images and a testing set of 36. After training, the network achieves an accuracy of 1. on both the training and the testing set. This shows that even though we train a small network on a toy dataset, it is able to learn significant features.

After quantising the network's weights, we export them using the dedicated script. We can then represent the network in Imandra:

```
let process_output (output: Z.t list) = match output with
| fst::(snd::[]) -> if fst < snd then Ok "Sad" else Ok "Happy"
| _ -> Error "error"

let model_raw input = layer_0 input >>= layer_1 >>= layer_2 >>= layer_4
  ;;
```

```
let model input = layer_0 input >>= layer_1 >>= layer_2 >>= layer_4 >>=
    process_output;;
```

The `process_output` function is a sort of argmax function to determine the network's classification of the input. With this representation, we can then verify several known robustness properties using the functions implemented in Section 4.3:

```
let epsilon = 1

(* checks if the given matrix's elements are binary *)
let is_binary m = let m' = Matrix.map (fun x -> x = 0 || x = 1) m in
let fold_true x = List.fold_left (&&) true x in
m' >>= (fun x -> Ok (fold_true (List.map fold_true x)))

let is_valid x = match x with
| x::[] -> Matrix.is_valid x && Matrix.dimensions x = (9, 9) &&
  is_binary x = (Ok true)
| _ -> false

(* verify classification robustness *)
verify (fun x -> Robustness.cr model input "Happy" epsilon ~constrnt:(
  is_valid x) x) [@@blast]

(* verify standard robustness *)
verify (fun x -> Robustness.sr model_raw input 1 epsilon ~constrnt:(
  is_valid x) x) [@@blast]

(* verify Lipschitz robustness *)
verify (fun x -> Robustness.lr model_raw input 2 epsilon ~constrnt:(
  is_valid x) x) [@@blast]

(* verify approximate classification robustness *)
verify (fun x -> Robustness.sr model_raw input 1 epsilon $\tilde{}
  $constrnt:(is_valid x) x) [@@blast]
```

We test the network's robustness around an input of each class, shown in Figure 5.1. The values of ϵ and parameters specific to each robustness definition are given in Table 5.2

5.1.3 Results and Interpretation

Table 5.2 shows the verification result for the verifications of all robustness definitions given in (Casadio et al., 2021) for our example CNN, around the happy and sad outputs

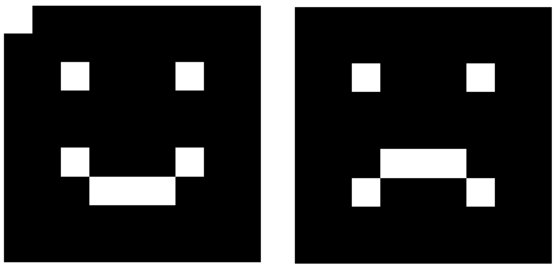


FIGURE 5.1: Inputs used to verify robustness.

	Happy		**Sad**	
Property	*Result*	*Time (s)*	*Result*	*Time (s)*
$CR(\epsilon)$	Refuted	96.36	Refuted	89.35
$SR(\epsilon, \delta)$	Proved	107.12	Proved	108.37
$LR(\epsilon, L)$	Proved	110.74	Proved	117.43
$ACR(\epsilon, \eta)$	Refuted	90.47	Proved	89.00

TABLE 5.2: CNN robustness verification results. The parameters are: $\epsilon = 1, \delta = 1, L = 2, \eta = 1$

shown in Figure .

We can see that all the properties verification were successful, meaning that they terminated and Imandra gave a definite ”proved” or ”refuted” result. In the case of a refutation, Imandra gives an executable counter-example. Listing 5.1 shows the result for the classification property of the sad input; we then execute the model with the counter-example as input in order to verify that it indeed classifies it as ”Happy”. The counter-example is visualised in Figure 5.2.

```
# verify (fun x -> Robustness.cr model input "Sad" epsilon ~constrnt:(
    is_valid x) x) [@@blast];;
  - : Z.t Matrix.matrix vector -> bool = <fun>
  counterexample from "blast" (after 89.35s, 5833 steps)
  Counterexample (after 5833 steps, 89.347s):
  let x : int Matrix.matrix list = [...]
  [] Conjecture refuted.
  module CX : sig val x : Z.t Matrix.matrix vector end
# CX.x;;
  - : Z.t Matrix.matrix vector =
  [[[0;0;0;0;0;0;0;0;0];[0;0;0;0;0;0;0;0;0];[0;0;1;0;0;0;1;0;0];
  [0;0;0;0;0;0;0;0;0];[0;0;0;0;0;0;0;0;0];[0;0;0;1;1;1;0;0;0];
  [0;0;0;0;0;1;0;0];[0;0;0;0;0;0;0;0;0];[0;0;0;0;0;0;0;0;0]]]
```

```
# model CX.x;;
  - : (string, string) result = Ok "Happy"
```

Listing 5.1: Imandra command line output for the verification of classification robustness around the sad input (lines starting with # are user inputs).

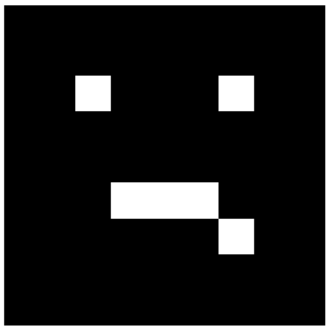


Figure 5.2: Visualisation of the counter-example generated by Imandra when verifying classification robustness for the sad example. We can see that it is valid (dimensions 9×9, binary values) and that its L0 distance with the original sad image (i.e. the number of pixels with different values) is 1.

Note that approximate classification robustness is supposed to be used with networks with a softmax output layer, and verify that the score for the expected class is greater than a fixed threshold set to 0.5 in the original paper (Fischer et al., 2019).

Since our library does not implement softmax layers and uses integers, it does not make sense as a property for our example; we chose 1 arbitrarily as a threshold value for the demonstration.

The conclusions of this experiment are two-fold. Firstly, the payoff of implementing a large general library for CNNs and matrices is that we are able to implement and verify different definitions of robustness on a CNN in very few lines of code, in a clear syntax. This is a positive sign of Imandra's ease of use as a verification tool. Secondly, we managed to experimentally confirm the suggestion by (Casadio et al., 2021) that verifying different definitions of robustness on the same network yields different results; this speaks for the importance of distinguishing between formal definitions of robustness, and for the future usability of our Imandra library that provides all these definitions in a generic way.

5.2 Investigating Novel Verification Properties with Imandra

Most of the verification and machine learning literature is concerned with various robustness properties of NNs (see to name a few (Katz et al., 2019), (Singh et al., 2019), (Huang et al., 2017)).

However, robustness as a property has limited applications, and owes its popularity mostly to its conceptual simplicity. Ideally, when we talk about imposing safety guarantees on NNs, we'd prefer to be able to reason at the level of features that have conceptual meaning to human users.

It would be tempting to conjecture that verification tools like Imandra may open the way for this novel style of verification, thanks to greater transparency and better mathematical clarity of their definitions and functions. This section is thus devoted to experiments that show how this novel methodology may work in practice.

The general idea is as follows: machine learning literature often alludes to filters in CNNs as constructions that bear the record of meaningful learned features(see Section 2.1.4.1). Informally, when one visualises the heatmaps of weights for the feature maps (as shown in Figure 5.3), one can notice certain shapes: e.g. a horizontal line, a diagonal line, a corner, etc. It is understood that perhaps the neural network learns a “definition” of the given object by detecting how those different kinds of lines are located in the given image.

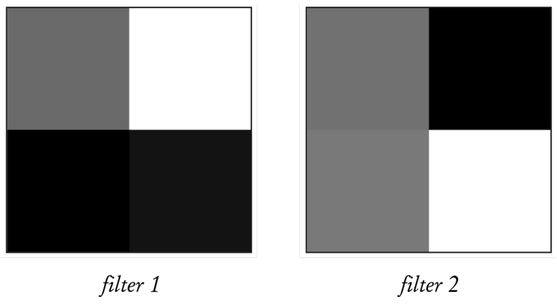


FIGURE 5.3: Weights heatmaps of the two $2\times2\times1$ filters of layer 0 of the CNN described in Section 5.1.2. Filter 1 presents a top-right corner pattern, filter 2 a top-left corner.

In this chapter we use our Imandra library to test this folklore hypothesis in the following two ways:

- We show how one can formally define shapes and lines for filters. We then use Imandra to formally analyse their role in image classification via assessing the "hot areas" in the pooling layer. We show that the features learned by a 100% accurate neural network may not in fact correspond to the most straightforward interpretation of the objects' features that a human would give.

In the future, one may prefer to use verification techniques to impose more intuitive, possibly user-defined, features to be present in neural network construction. With that in mind,

- We show how the features learned by a neural network can be substituted by user-defined and conceptually more meaningful features. Moreover, we show that:
  - the resulting networks do not lose accuracy after the intervention (assuming we can re-train fully-connected parts of the network),
  - they are more likely to yield the verification properties of interest, thanks to improved conceptual understanding of the structural properties of the given neural network.

Although we do this for our small running example of happy and sad faces, we believe that the same method could be replicated for data sets of any size.

5.2.1 Formalising Shapes of Filters and Feature Map Patterns in Imandra

In this section, we use the network described in Section 5.1.2, represented in Figure 2.7 to test our hypothesis, with its filters *filter 1* and *filter 2* represented in Figure 5.3. We note that these patterns were learned by the CNN automatically. So, our task here is to simply understand and conceptualise the properties of these patterns and this CNN.

By looking at filter 1 and comparing it to the images in our dataset, we conjecture that the network uses it to identify images if matches areas in the bottom left part of the image (see Figure 5.4).

To express this conjecture, we first implement functions that check patterns on 2×2 matrices (see Listing 5.2).

```
let is_approx a b delta = let abs x = if x <. 0. then (-1. *. x) else x
  in
Res.lift2 (fun a' b' -> abs(a' -. b') <=. delta) a b;;


(* check if there is a diagonal from top left to bottom right *)
```

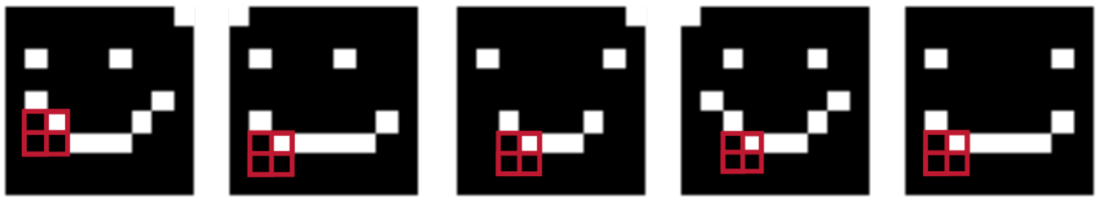


FIGURE 5.4: We see that for "happy" images, filter 1 (top-left corner) will be detected in the bottom left part of the image.

```
let is_left_diagonal m =
let is_valid = Matrix.dimensions m =  (2, 2) in
let tl = Matrix.nth (0, 0) m in
let tr = Matrix.nth (0, 1) m in
let bl = Matrix.nth (1, 0) m in
let br = Matrix.nth (1, 1) m in
let delta = 0.1 in
if is_valid && (is_approx tl br delta) &&? (bl <? tl) &&? (tr <? tl) =
  (Ok true)
then (Ok true)
else (Ok false)

(* check if the top right corner has a higher value than the other
  elements *)
let is_tr_corner m =
let is_valid = Matrix.dimensions m =  (2, 2) in
let tl = Matrix.nth (0, 0) m in
let tr = Matrix.nth (0, 1) m in
let bl = Matrix.nth (1, 0) m in
let br = Matrix.nth (1, 1) m in
if is_valid && (bl <? tr) &&? (tl <? tr) &&? (br <? tr) = (Ok true)
then (Ok true)
else (Ok false)
```

LISTING 5.2: Examples of pattern definition in Imandra. Refer to Appendix A for the complete formalisation

Our library's matrix implementation helps us formalise these properties in IML easily. We also defined useful functions: `is_approx`, which compares the equality of two values within a given precision, and comparison operators for `Result` type (`<?, ||?, &&?`).

We then conjecture that the filter1 matching happy inputs result in specific patterns in the feature map, which in turn result in specific patterns in the pooling layer (see Figure 5.4). It is more interesting to reason about the pooling layer, as it is smaller and only keeps the feature map's most significant values.

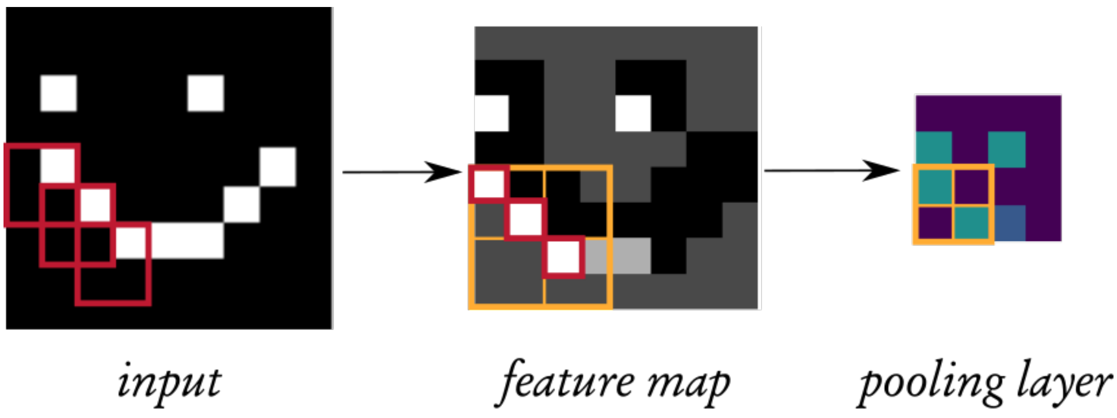


FIGURE 5.5: .

By combining the detection of different patterns in the pooling layer, we can then express a property P_1 that applies to the pooling layer for all happy inputs. We implement it using the pattern functions from the previous Listing:

```
let happy_properties (l: (real Matrix.matrix list)) =
  let f1 = Vec.hd l in

  (* f1' is 2x2 submatrix in the bottom left corner of the pooling layer
   for filter 1*)
  let f1' = Matrix.sub_matrix f1 (2, 0) (2, 2) in

  (* f1' presents a left-vertical line pattern *)
  let left_vertical = f1' >>= is_left_vertical in

  (* f1' presents a bottom-horizontal line pattern *)
  let bottom_horizontal = f1' >>= is_bottom_horizontal in

  (* f1' presents a left-diagonal line pattern *)
  let left_diagonal = f1' >>= is_left_diagonal in

  (* f1' presents a top-left corner pattern *)
  let tl_corner = f1' >>= is_tl_corner in

  (* f1' presents a negative top-right corner pattern *)
  let bl_angle = f1' >>= is_bl_angle in
  left_vertical ||? bottom_horizontal ||? left_diagonal ||? tl_corner ||?
    bl_angle
```

LISTING 5.3: Definition of P1 in IML

We gain in explainability: when our network classifies an input as happy, if this property holds we know that the network has based its classification on a meaningful feature.

We gave meaning to the pattern of an automatically learned filter; we managed to formalise property P_1 can explain why images are classified as happy.

However, P_1 is quite convoluted and not very meaningful for a human. A solution is to impose filters that represent meaningful features from the start, which we try in the next section.

5.2.2 Imposing Properties of Filters in Imandra

We define filters that represent meaningful features, namely diagonal lines, shown in Figure 5.6.

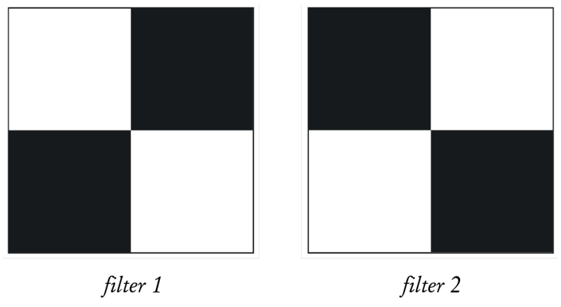


FIGURE 5.6: Weights heatmaps of the two $2 \times 2 \times 1$ filters imposed on a re-trained CNN.

It is easy to conjecture that by detecting the positions of diagonal lines that form the ”smile” and ”frown” part of the images, these filters can be used to differentiate happy and sad faces (see Figure 5.7).

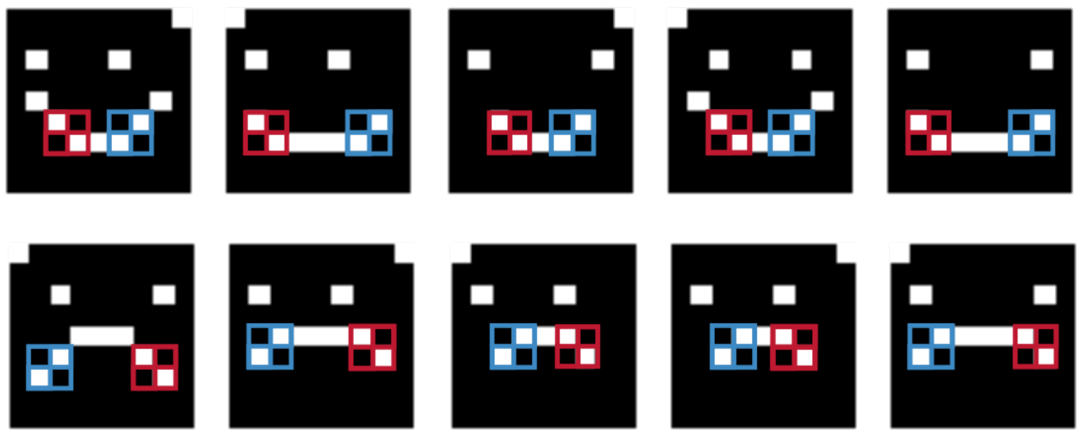


FIGURE 5.7: The corners of the image where left-diagonal (in red) and right-diagonal (in blue) filters are detected.

We imposed two filters corresponding to the diagonal features (see Figure 5.6). We then re-trained the network's final fully-connected layer. The re-trained network reached an accuracy of 100% on both training and testing sets, which confirms our intuition that the diagonal features are discriminating between the two classes.

We formalise our intuition by checking for happy images if the pooling layer shows a perfect match of filter 1 in the bottom left of the image and for filter 2 in the bottom right. This property P_2 is formalised in IML as:

```
(* function to check if the filter has matched at least once in the given
     region (for binary filters) *)
let at_least_one_match value filter =
  let is_valid = Matrix.dimensions filter =  (2, 2) in
  let matched x = x >>= (fun x -> Ok (x = value)) in
  let tl = Matrix.nth (0, 0) filter in
  let tr = Matrix.nth (0, 1) filter in
  let bl = Matrix.nth (1, 0) filter in
  let br = Matrix.nth (1, 1) filter in
  if is_valid && (matched tl) ||? (matched tr) ||? (matched bl) ||? (
   matched br) = (Ok true)
    then (Ok true)
    else (Ok false)


let happy_binary_properties l =
  let f1 = Vec.nth 0 l in
  let f2 = Vec.nth 1 l in
  (* bottom left corner of pooling output for filter 1 *)
  let f1' = f1 >>= (fun x -> Matrix.sub_matrix x (2, 0) (2, 2)) in
  (* bottom right corner of pooling output for filter 2 *)
  let f2' = f2 >>= (fun x -> Matrix.sub_matrix x (2, 2) (2, 2)) in
  let bl = f1' >>= (at_least_one_match 2.) in
  let br = f2' >>= (at_least_one_match 2.) in
  bl &&? br
```

This property held for all happy instances of the training set, and a symmetric property held as well for all sad instances.

5.2.3 Lessons Learnt

Our library allowed us to explore a novel verification property. We were able to conceptualise a CNN's filters' meaning by formalising properties on the network's pooling layers, which helps improving explainability for our CNN. This flexibility speaks in favour of Imandra's suitability for CNN verification and explainability.

If we compare P_1 and P_2, we notice that P_2, which results from a network with imposed filters, is more meaningful to a human observer and easier to formalise in IML. The reason is that features learned by the network do not have intrinsic meaning to an observer, so it is harder to reason about them. Moreover, they do not have binary values, so analysing pooling layer patterns to determine properties is difficult. Hence, greater explainability might come from future tools that let verification engineers impose filters they think are relevant to the problem domain to NNs before training them.

The next step in this exploration is to turn the novel explainability properties into formal verification. For instance, if we expressed some constraints on the shape of a face in our dataset, we could formally prove with Imandra's theorem proving that property P_2 holds for all valid faces classified as happy by our network. However, due to lack of time, we weren't able to implement this part and leave it to future work.

5.3 Testing Against Existing Benchmark

Scalability to larger networks is a known issue for verification of larger NNs with SMT solvers (Kokke et al., 2020). Nonetheless, we tried to run a property verification from the ACAS Xu benchmark described in Section 3.2. The model we represented is a quantised version of the network used in (Imandra, 2019); because of the Blast strategy's limitation, we are not able to test our library against an exact existing benchmark.

We try to prove property ϕ_1 defined in (Katz et al., 2017): "If the intruder is distant and is significantly slower than the ownship, the score of a COC advisory will always be below a certain fixed threshold". It is expressed in IML as:

```
let is_valid input = match input with
| (dist::angle::angle_int::vown::vint::[]) -> -3 <= angle && angle <= 3
   &&     (* angles are in [-pi, pi] *)
-3 <= angle_int && angle_int <= 3 &&
dist >= 0 && vown >= 0 (* speed and distances are positive *)
| _ -> false

let condition1 input = match input with
| (dist::angle::angle_int::vown::vint::[]) ->
(dist <= 55948) && (vown >= 1145) && (vint <= 60)
| _ -> false

let property1 input = let output = model input in
let coc = Res.flatten (Res.map (Vec.nth 0) output) in
coc <=? 1500
```

```
verify (fun x -> is_valid x && condition1 x ==> property1 x) [@@blast]
```

Listing 5.4: Expression of a property from the ACAS Xu benchmark in IML

With 300 nodes, this network is too large for Blast to verify the property with our library. Our library allows to express easily models and properties from the ACAS Xu benchmark, but the properties cannot be verified. Had we had more time, we could have tried network distillation techniques to slim the NN down to a verifiable size for our library; we leave that to future work.

Chapter 6

Conclusions

In this section, we fist discuss what was learned from this project from a personal perspective, we then summarise this report's results and draw conclusions from them, and finally, we suggest possible future research.

6.1 Reflection

In this section, I will discuss the main challenges and lessons learned from this project.

NN verification is a relatively new and yet already vast research subject. It spans multiple disciplines, like computer science, data science, and formal logic. Being new to the field, assimilating material from these diverse sources in the first months of this project was daunting, but ended up being rewarding.

Having no previous experience with theorem provers in general and Imandra in particular, I learned how to use them for this project. Even though I had some experience in functional programming before, I also had to learn OCaml which proved to be a tedious task due to its peculiar syntax. Thankfully as for all new programming languages, regular practice helped me get accustomed to the language and its quirks.

Convolutional neural networks' implementation proved to be the most effort-intensive task in this project. It presented challenges both on a research level, namely CNN formalisation, and on an engineering level, in designing the library with IML's specificities taken into account.

6.2 Results

We managed to achieve all the objectives defined in this report.

O1. We created a library that formalises NNs in IML. Using standard functional language constructs, like the built-in list data type, we were able to implement a general and extendable matrix library. This library allowed us to formalise fully connected neural networks in a more flexible way than the existing library did.

O2. The matrix library allowed us to implement a novel convolutional neural network library in IML. It allowed us to formalise CNNs in Imandra's theorem prover, and thus to verify properties of CNNs.

O3. We created a script that converts NN's learned weights from a serialised representation to a valid IML file that can be read by Imandra, in order to verify properties of existing NNs.

O4. We evaluated our library's ability to formally verify CNN properties using Imandra's SMT-solving proof strategy. Using an artificial recognition dataset, we were able to formalise and verify robustness properties of a CNN around multiple inputs. IML's expressivity allowed us to easily define multiple definitions of robustness and to try and verify them.

O5. Using our library, we investigated novel CNN properties. In particular, we made an attempt at formalising and checking conceptual features learnt by a neural network, as well as imposing user-defined meaningful features during training. This work will hopefully inform further research into CNN verification.

6.3 Future Works

A number of small changes can be implemented to make our CNN library clearer, more flexible and more user-friendly:

- Use type parameters to generalise our matrix definition. This would allow real number and quantised versions of our library to be merged, and make our library more easily extensible to support new data types (for instance booleans for binarised NNs).
- Build a richer matrix datatype to enrich matrices' representation, for instance with their dimensions. This would make matrix dimension checking operations faster, improving our library's performance.

- Create a library of lemmas and axioms about our matrices and functions. Imandra could then use them in its proofs.
- At the moment we use an intermediate Python representation in order to convert a serialised NN into IML. One could implement a script in OCaml/IML that directly converts a serialised NN (for instance from an ONNX file) into an IML representation.

Time limitations forced us to leave some paths unexplored, both in the domain of Imandra's abilities to NN verification and the formulation of novel verification methods.

We have seen that our library's application to natural datasets like the ACAS Xu problem does not give a satisfactory answer. We suggest studying how distillation techniques could be used in order to first create a new network of reduced size and complexity but with similar performance, making it possible to verify its properties with Imandra.

In Section 5.2, we proposed a new explanation property. We believe it is worth investigating further; notably, it could be turned into a formal verification property by expressing the property into Imandra's theorem prover.

Finally, Probabilistic verification is a subject of increasing interest and Imandra has already proved capable of probabilistic reasoning. Hence, we believe that Imandra's potential application to probabilistic programming and probabilistic verification of NNs is worth exploring.

Bibliography

Attala, Z., Cavalcanti, A., & Woodcock, J. (2020, July 19). A comparison of neural network tools for the verification of linear specifications of ReLU networks, In *Proc. FoMLAS2020*. FoMLAS'2020, Los Angeles, USA.

Bastani, O., Ioannou, Y., Lampropoulos, L., Vytiniotis, D., Nori, A. V., & Criminisi, A. (2016). Measuring neural net robustness with constraints (D. D. Lee, M. Sugiyama, U. v. Luxburg, I. Guyon, & R. Garnett, Eds.). In D. D. Lee, M. Sugiyama, U. v. Luxburg, I. Guyon, & R. Garnett (Eds.), *Advances in neural information processing systems 29: Annual conference on neural information processing systems 2016, december 5-10, 2016, barcelona, spain*. https://proceedings.neurips.cc/paper/2016/hash/980ecd059122ce2e50136bda65c25e07-Abstract.html

Bojarski, M., Testa, D. D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., & Zieba, K. (2016). End to end learning for self-driving cars [_eprint: 1604.07316]. *CoRR*, *abs/1604.07316*. http://arxiv.org/abs/1604.07316

Bunel, R. R., Turkaslan, I., Torr, P., Kohli, P., & Mudigonda, P. K. (2018). A unified view of piecewise linear neural network verification (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, & R. Garnett, Eds.). In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, & R. Garnett (Eds.), *Advances in neural information processing systems*, Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/be53d253d6bc3258a8160556dda3e9b2-Paper.pdf

Cardelli, L., Kwiatkowska, M., Laurenti, L., & Patane, A. (2019). Robustness guarantees for bayesian inference with gaussian processes, In *The thirty-third AAAI conference on artificial intelligence, AAAI 2019, the thirty-first innovative applications of artificial intelligence conference, IAAI 2019, the ninth AAAI symposium on educational advances in artificial intelligence, EAAI 2019, honolulu, hawaii, USA, january 27 - february 1, 2019*, AAAI Press. https://doi.org/10.1609/aaai.v33i01.33017759

Carlini, N., & Wagner, D. (2017, May). Towards evaluating the robustness of neural networks, In *2017 IEEE symposium on security and privacy (SP)*. 2017 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, IEEE. https://doi.org/10.1109/SP.2017.49

Casadio, M., Daggitt, M. L., Komendantskaya, E., Kokke, W., Kienitz, D., & Stewart, R. (2021). Property-driven training: All you (n)ever wanted to know about [_eprint: 2104.01396]. *CoRR*, *abs/2104.01396*. https://arxiv.org/abs/2104.01396

Elboher, Y. Y., Gottschlich, J., & Katz, G. (2020). An abstraction-based framework for neural network verification (S. K. Lahiri & C. Wang, Eds.). In S. K. Lahiri & C. Wang (Eds.), *Computer aided verification - 32nd international conference, CAV 2020, los angeles, CA, USA, july 21-24, 2020, proceedings, part i*, Springer. https://doi.org/10.1007/978-3-030-53288-8_3

Fischer, M., Balunovic, M., Drachsler-Cohen, D., Gehr, T., Zhang, C., & Vechev, M. T. (2019). DL2: Training and querying neural networks with logic (K. Chaudhuri & R. Salakhutdinov, Eds.). In K. Chaudhuri & R. Salakhutdinov (Eds.), *Proceedings of the 36th international conference on machine learning, ICML 2019, 9-15 june 2019, long beach, california, USA*, PMLR. http://proceedings.mlr.press/v97/fischer19a.html

Fisher, K., Launchbury, J., & Richards, R. (2017). The HACMS program: Using formal methods to eliminate exploitable bugs. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, *375*(2104), 20150401. https://doi.org/10.1098/rsta.2015.0401

Fun2model. (2019). *FUN2model*. Retrieved April 1, 2021, from http://www.fun2model.org/

Goodfellow, I. J., Shlens, J., & Szegedy, C. (2015). Explaining and harnessing adversarial examples (Y. Bengio & Y. LeCun, Eds.). In Y. Bengio & Y. LeCun (Eds.), *3rd international conference on learning representations, ICLR 2015, san diego, CA, USA, may 7-9, 2015, conference track proceedings*. http://arxiv.org/abs/1412.6572

Gopinath, D., Converse, H., Pasareanu, C. S., & Taly, A. (2019). Property inference for deep neural networks, In *34th IEEE/ACM international conference on automated software engineering, ASE 2019, san diego, CA, USA, november 11-15, 2019*, IEEE. https://doi.org/10.1109/ASE.2019.00079

Guidotti, D., Pulina, L., & Tacchella, A. (2020). NeVer 2.0: Learning, verification and repair of deep neural networks [_eprint: 2011.09933]. *CoRR*, *abs/2011.09933*. https://arxiv.org/abs/2011.09933

Haykin, S. S. (2009). *Neural networks and learning machines* (Third). Upper Saddle River, NJ, Pearson Education.

Hinton, A., Kwiatkowska, M., Norman, G., & Parker, D. (2006). PRISM: A tool for automatic verification of probabilistic systems (H. Hermanns & J. Palsberg, Eds.). In H. Hermanns & J. Palsberg (Eds.), *Tools and algorithms for the construction and analysis of systems*, Berlin, Heidelberg, Springer. https://doi.org/10.1007/11691372_29

Huang, X., Kwiatkowska, M., Wang, S., & Wu, M. (2017). Safety verification of deep neural networks (R. Majumdar & V. Kuncak, Eds.). In R. Majumdar & V. Kuncak (Eds.), *Computer aided verification - 29th international conference, CAV 2017, heidelberg, germany, july 24-28, 2017, proceedings, part i*, Springer. https://doi.org/10.1007/978-3-319-63387-9_1

Imandra. (2019). *Nn2iml*. Retrieved August 4, 2021, from https://github.com/AestheticIntegration/imandra-stats-experiments

Imandra. (2021a). *Probabilistic reasoning in OCaml* [Imandra documentation]. Retrieved April 7, 2021, from https://docs.imandra.ai/imandra-docs/notebooks/probabilistic-reasoning-in-ocaml/

Imandra. (2021b). *Proving program termination with imandra* [Documentation]. Retrieved August 6, 2021, from https://docs.imandra.ai/imandra-docs/notebooks/proving-program-termination/

Imandra. (2021c). *Verification waterfall* [Imandra documentation]. Retrieved August 6, 2021, from https://docs.imandra.ai/imandra-docs/notebooks/verification-waterfall/

Johnsen, E. B. (2021). *Formal methods*. Retrieved April 8, 2021, from http://www.fmeurope.org/formalmethods/

Julian, K. D., Lopez, J., Brush, J. S., Owen, M. P., & Kochenderfer, M. J. (2016, September). Policy compression for aircraft collision avoidance systems [ISSN: 2155-7209], In *2016 IEEE/AIAA 35th digital avionics systems conference (DASC)*. 2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC). ISSN: 2155-7209. https://doi.org/10.1109/DASC.2016.7778091

Katz, G., Barrett, C. W., Dill, D. L., Julian, K., & Kochenderfer, M. J. (2017). Reluplex: An efficient SMT solver for verifying deep neural networks (R. Majumdar & V. Kuncak, Eds.). In R. Majumdar & V. Kuncak (Eds.), *Computer aided verification - 29th international conference, CAV 2017, heidelberg, germany, july 24-28, 2017, proceedings, part i*, Springer. https://doi.org/10.1007/978-3-319-63387-9_5

Katz, G., Huang, D. A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., Dill, D. L., Kochenderfer, M. J., & Barrett, C. (2019). The marabou framework for verification and analysis of deep neural networks [Series Title: Lecture Notes in Computer Science]. In I. Dillig & S. Tasiran (Eds.), *Computer aided verification* (pp. 443–452). Series Title: Lecture Notes in Computer

Science. Cham, Springer International Publishing. https://doi.org/10.1007/978-3-030-25540-4_26

Kim, E., Gopinath, D., Pasareanu, C. S., & Seshia, S. A. (2019). A programmatic and semantic approach to explaining and DebuggingNeural network based object detectors [_eprint: 1912.00289]. *CoRR*, *abs/1912.00289*. http://arxiv.org/abs/1912.00289

Kokke, W. (2020, December 7). *Wenkokke/sapphire* [original-date: 2020-02-19T21:46:47Z]. Retrieved April 1, 2021, from https://github.com/wenkokke/sapphire

Kokke, W., Komendantskaya, E., Kienitz, D., Atkey, R., & Aspinall, D. (2020). Neural networks, secure by construction: An exploration of refinement types [Series Title: Lecture Notes in Computer Science]. In *Programming languages and systems* (pp. 67–85). Series Title: Lecture Notes in Computer Science. Cham, Springer International Publishing. https://doi.org/10.1007/978-3-030-64437-6_4

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks (F. Pereira, C. J. C. Burges, L. Bottou, & K. Q. Weinberger, Eds.). In F. Pereira, C. J. C. Burges, L. Bottou, & K. Q. Weinberger (Eds.), *Advances in neural information processing systems*, Curran Associates, Inc. https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, *1*(4), 541–551. https://doi.org/10.1162/neco.1989.1.4.541

M. Wicker, X. H., & Kwiatkowska, M. (2018). Feature-guided black-box safety testing of deep neural networks, In *Proc. 24th international conference on tools and algorithms for the construction and analysis of systems (TACAS'18)*.

Madry, A., Makelov, A., Schmidt, L., Tsipras, D., & Vladu, A. (2018). Towards deep learning models resistant to adversarial attacks, In *6th international conference on learning representations, ICLR 2018, vancouver, BC, canada, april 30 - may 3, 2018, conference track proceedings*, OpenReview.net. https://openreview.net/forum?id=rJzIBfZAb

Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., & Deardeuff, M. (2015). How amazon web services uses formal methods. *Communications of the ACM*, *58*(4), 66–73. https://doi.org/10.1145/2699417

Owl. (2021, August 6). *Mask r-CNN* [original-date: 2018-08-21T16:37:12Z]. Owl - OCaml Scientific Computing. Retrieved August 15, 2021, from https://github.com/owlbarn/owl_mask_rcnn

Passmore, G., Cruanes, S., Ignatovich, D., Aitken, D., Bray, M., Kagan, E., Kanishev, K., Maclean, E., & Mometto, N. (2020). The imandra automated reasoning system

(system description) (N. Peltier & V. Sofronie-Stokkermans, Eds.). In N. Peltier & V. Sofronie-Stokkermans (Eds.), *Automated reasoning*, Cham, Springer International Publishing. https://doi.org/10.1007/978-3-030-51054-1_30

POPL. (2021). *Program*. Retrieved April 8, 2021, from https://popl21.sigplan.org/program/program-POPL-2021

Ribeiro, M. T., Singh, S., & Guestrin, C. (2016). "why should i trust you?": Explaining the predictions of any classifier (B. Krishnapuram, M. Shah, A. J. Smola, C. C. Aggarwal, D. Shen, & R. Rastogi, Eds.). In B. Krishnapuram, M. Shah, A. J. Smola, C. C. Aggarwal, D. Shen, & R. Rastogi (Eds.), *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining, san francisco, CA, USA, august 13-17, 2016*, ACM. https://doi.org/10.1145/2939672.2939778

Rushby, J., & Whitehurst, R. A. (1989). *Formal verification of AI software*. National Aeronautics; Space Administration, Langley Research Center.

Shorten, C., & Khoshgoftaar, T. M. (2019). A survey on image data augmentation for deep learning [Publisher: Springer Science and Business Media LLC]. *Journal of Big Data*, *6*(1). https://doi.org/10.1186/s40537-019-0197-0

Singh, G., Gehr, T., Püschel, M., & Vechev, M. (2019). An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, *3*, 1–30. https://doi.org/10.1145/3290354

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I. J., & Fergus, R. (2014). Intriguing properties of neural networks (Y. Bengio & Y. LeCun, Eds.). In Y. Bengio & Y. LeCun (Eds.), *2nd international conference on learning representations, ICLR 2014, banff, AB, canada, april 14-16, 2014, conference track proceedings*. http://arxiv.org/abs/1312.6199

Tassarotti, J., Vajjha, K., Banerjee, A., & Tristan, J.-B. (2021). A formal proof of PAC learnability for decision stumps, In *CPP '21: 10th ACM SIGPLAN international conference on certified programs and proofs, virtual event, denmark, january 17-19, 2021*. https://doi.org/10.1145/3437992.3439917

Urban, C., Christakis, M., Wüstholz, V., & Zhang, F. (2020). Perfectly parallel fairness certification of neural networks. *Proceedings of the ACM on Programming Languages*, *4*, 1–30. https://doi.org/10.1145/3428253

Wang, S., Pei, K., Whitehouse, J., Yang, J., & Jana, S. (2018). Formal security analysis of neural networks using symbolic intervals (W. Enck & A. P. Felt, Eds.). In W. Enck & A. P. Felt (Eds.), *27th USENIX security symposium, USENIX security 2018, baltimore, MD, USA, august 15-17, 2018*, USENIX Association. https://www.usenix.org/conference/usenixsecurity18/presentation/wang-shiqi

Appendix A

Patterns' Implementation

The full implementation for the patterns on 2×2 matrices discussed in Section 5.2 and implemented in the code is defined in the following Listing.

```
let is_approx a b delta = let abs x = if x <. 0. then (-1. *. x) else x
    in
Res.lift2 (fun a' b' -> abs(a' -. b') <=. delta) a b;;

let lt x y = Res.bind2 x y (fun a b -> Ok (a <. b));;
let (<?) = lt;;

let res_or x y = Res.bind2 x y (fun a b -> Ok (a || b));;
let (||?) = res_or;;

let res_and x y = Res.bind2 x y (fun a b -> Ok (a && b));;
let (&&?) = res_and;;

(* functions to test patterns on 2x2 matrices *)

(* check if there is a diagonal from top left to bottom right *)
let is_left_diagonal m =
let is_valid = Matrix.dimensions m =  (2, 2) in
let tl = Matrix.nth (0, 0) m in
let tr = Matrix.nth (0, 1) m in
let bl = Matrix.nth (1, 0) m in
let br = Matrix.nth (1, 1) m in
let delta = 0.1 in
if is_valid && (is_approx tl br delta) &&? (bl <? tl) &&? (tr <? tl) = (
    Ok true)
then (Ok true)
else (Ok false)
```

```
;;

(* check if there is a diagonal from top right to bottom left *)
let is_right_diagonal m =
let is_valid = Matrix.dimensions m =  (2, 2) in
let tl = Matrix.nth (0, 0) m in
let tr = Matrix.nth (0, 1) m in
let bl = Matrix.nth (1, 0) m in
let br = Matrix.nth (1, 1) m in
let delta = 0.1 in
if is_valid && (is_approx tr bl delta) &&? (br <? tr) &&? (tl <? tr) = (
    Ok true)
then (Ok true)
else (Ok false)
;;

let is_diagonal m =
(is_left_diagonal m) ||? (is_right_diagonal m)
;;

(* check if there is a vertical line on the left of the square *)
let is_left_vertical m =
let is_valid = Matrix.dimensions m =  (2, 2) in
let tl = Matrix.nth (0, 0) m in
let tr = Matrix.nth (0, 1) m in
let bl = Matrix.nth (1, 0) m in
let br = Matrix.nth (1, 1) m in
let delta = 0.1 in
if is_valid && (is_approx tl bl delta) &&? (tr <? tl) &&? (br <? tl) = (
    Ok true)
then (Ok true)
else (Ok false)
;;

(* check if there is a horizontal line at the bottom of square *)
let is_bottom_horizontal m =
let is_valid = Matrix.dimensions m =  (2, 2) in
let tl = Matrix.nth (0, 0) m in
let tr = Matrix.nth (0, 1) m in
let bl = Matrix.nth (1, 0) m in
let br = Matrix.nth (1, 1) m in
let delta = 0.1 in
let lt x y = Res.bind2 x y (fun a b -> Ok (a <. b)) in
if is_valid && (is_approx bl br delta) &&? (tl <? bl) &&? (tr <? bl) = (
    Ok true)
then (Ok true)
else (Ok false)
;;
```

```
(* check if there is a horizontal line at the top of square *)
let is_top_horizontal m =
let is_valid = Matrix.dimensions m =  (2, 2) in
let tl = Matrix.nth (0, 0) m in
let tr = Matrix.nth (0, 1) m in
let bl = Matrix.nth (1, 0) m in
let br = Matrix.nth (1, 1) m in
let delta = 0.1 in
let lt x y = Res.bind2 x y (fun a b -> Ok (a <. b)) in
if is_valid && (is_approx tl tr delta) &&? (br <? tl) &&? (br <? tr) = (
    Ok true)
then (Ok true)
else (Ok false)
;;

(* check if the top left corner is higher *)
let is_tl_corner m =
let is_valid = Matrix.dimensions m =  (2, 2) in
let tl = Matrix.nth (0, 0) m in
let tr = Matrix.nth (0, 1) m in
let bl = Matrix.nth (1, 0) m in
let br = Matrix.nth (1, 1) m in
if is_valid && (bl <? tl) &&? (tr <? tl) &&? (br <? tl) = (Ok true)
then (Ok true)
else (Ok false)
;;

(* check if the top right corner is higher *)
let is_tr_corner m =
let is_valid = Matrix.dimensions m =  (2, 2) in
let tl = Matrix.nth (0, 0) m in
let tr = Matrix.nth (0, 1) m in
let bl = Matrix.nth (1, 0) m in
let br = Matrix.nth (1, 1) m in
if is_valid && (bl <? tr) &&? (tl <? tr) &&? (br <? tr) = (Ok true)
then (Ok true)
else (Ok false)
;;

(* check if the bottom left corner is higher *)
let is_bl_corner m =
let is_valid = Matrix.dimensions m =  (2, 2) in
let tl = Matrix.nth (0, 0) m in
let tr = Matrix.nth (0, 1) m in
let bl = Matrix.nth (1, 0) m in
let br = Matrix.nth (1, 1) m in
if is_valid && (tl <? bl) &&? (tr <? bl) &&? (br <? bl) = (Ok true)
```

```
then (Ok true)
else (Ok false)
;;

let is_bl_angle m =
let is_valid = Matrix.dimensions m =  (2, 2) in
let tl = Matrix.nth (0, 0) m in
let tr = Matrix.nth (0, 1) m in
let bl = Matrix.nth (1, 0) m in
let br = Matrix.nth (1, 1) m in
let delta = 0.1 in
if is_valid && (is_approx bl tl delta) &&? (is_approx br bl delta) &&? (
    tr <? bl) = (Ok true)
then (Ok true)
else (Ok false)
;;

(* function to check if the filter has matched at least once in the given
     region (for binary filters) *)
let at_least_one_match value filter =
let is_valid = Matrix.dimensions filter =  (2, 2) in
let matched x = x >>= (fun x -> Ok (x = value)) in
let tl = Matrix.nth (0, 0) filter in
let tr = Matrix.nth (0, 1) filter in
let bl = Matrix.nth (1, 0) filter in
let br = Matrix.nth (1, 1) filter in
if is_valid && (matched tl) ||? (matched tr) ||? (matched bl) ||? (
    matched br) = (Ok true)
then (Ok true)
else (Ok false)
;;
```

Appendix B

Professional, Legal, Ethical and Social Issues

B.1 Professional Issues

Professional issues that might arise during this project are linked with ”responsible research and innovation”. In order to make this project reproducible, all the data, environment and results of the experiments will be transparently reported. The produced software will be made available through a permissive open-source license to allow its use by as many people as possible.

B.2 Legal Issues

Legal issues might arise in this project regarding licensing of code. Imandra Core's code is closed source, but the Imandra-prelude library, which is included with OCaml code extracted from IML, contains code from libraries under the FreeBSD license. The license for these library will have to be included in the library deliverable.

A suitable license will have to be chosen for the library. In order to allow for best reusability of the code, it will most likely be published under a permissive open-source license like the MIT license.

B.3 Ethical Issues

As no sensitive data will be used and no experiments involving users will be conducted, no ethical issues are expected to arise from this project.

B.4 Social Issues

This project treats of a type of AI safety. This poses the question of the responsibility of autonomous systems: who is responsible when a self-driving car has an accident? The passenger in the car like it is the case now? The programmer who wrote the collision detection algorithm? Or the verification tool which failed to detect a possible adversarial situation? Even though this question is not directly related to our project, it is posed by the concept of AI verification itself.

Appendix C

Project Plan

C.1 Risk Assessment

This project mixes research and software development. As such, planning for tasks' start dates and durations – which even in regular software development is somewhat of a dark art – is broadly indicative at best.

C.2 Work Breakdown Structure

To follow the Work Breakdown Structure guideline, the lowest level task are no longer than a single reporting period, i.e. one week.

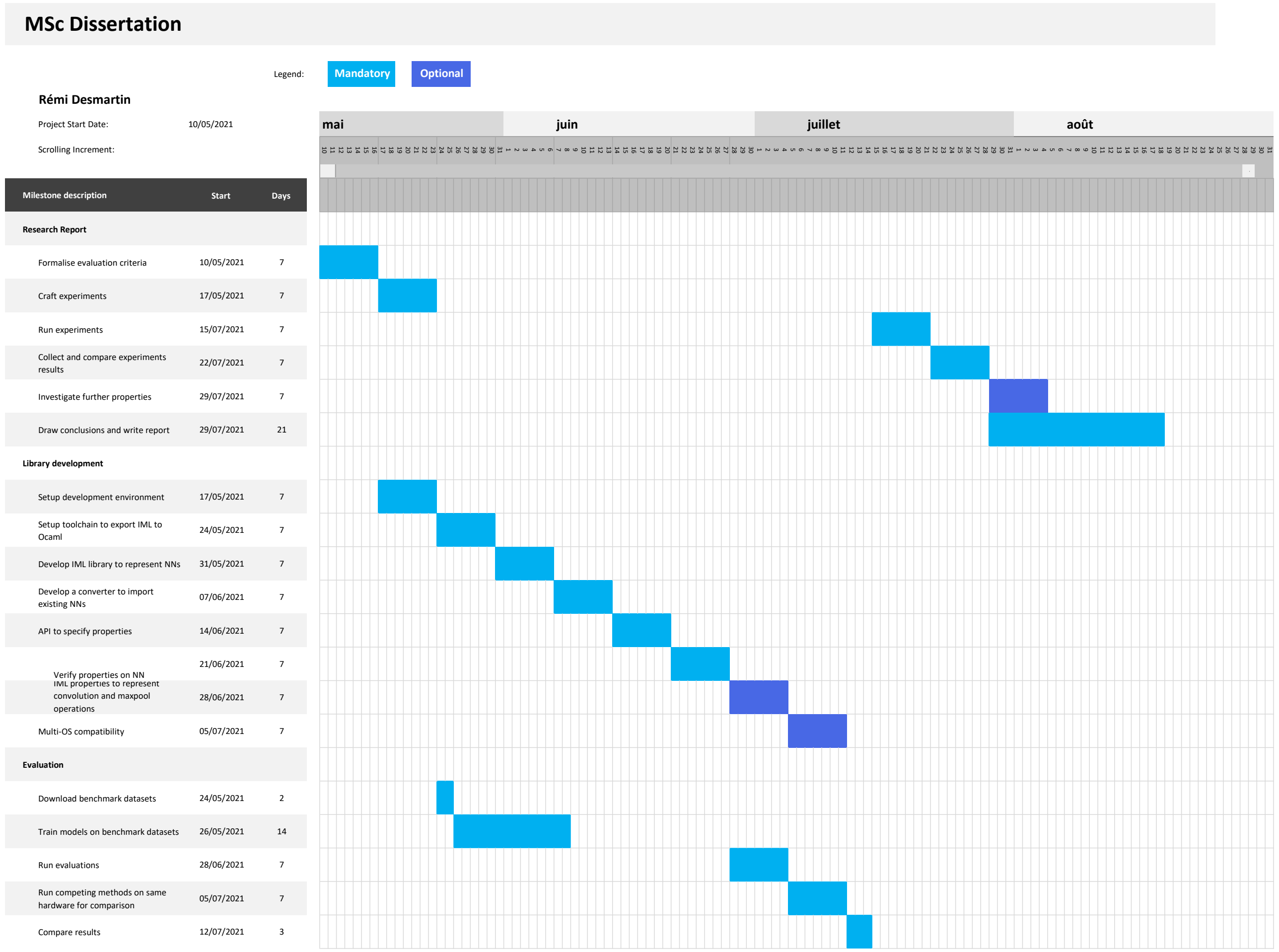
1. Research report
   - (a) Formalize evaluation criteria.
   - (b) Craft experiments.
   - (c) Run experiments.
   - (d) Collect and compare experiments' results.
   - (e) Investigate further properties.
   - (f) Draw conclusions and write report.
2. Library development.
   - (a) Setup development environment.
   - (b) Setup toolchain to export IML to OCaml.

   (c) Develop IML library to represent NNs.
   (d) Develop a converter to import existing NNs.
   (e) Develop an API to specify properties to verify
   (f) Verify properties on NNs
   (g) Develop IML library to represent convolutional and maxpool layers.
   (h) Either test on multiple operating systems or setup Docker image for portability.

3. Evaluation
   (a) Download benchmark datasets.
   (b) Train models on benchmark datasets if necessary.
   (c) Run evaluations.
   (d) Run competing methods on same hardware.
   (e) Compare results.

C.3 Gantt Chart

Figure C.1 shows a Gantt chart for this project. From past experience, the report writing task is planned to last three weeks, representing two major revisions after feedback.



MSc Dissertation

Legend: Mandatory Optional

Rémi Desmartin

Project Start Date: 10/05/2021

Scrolling Increment:

Milestone description	Start	Days
Research Report		
Formalise evaluation criteria	10/05/2021	7
Craft experiments	17/05/2021	7
Run experiments	15/07/2021	7
Collect and compare experiments results	22/07/2021	7
Investigate further properties	29/07/2021	7
Draw conclusions and write report	29/07/2021	21
Library development		
Setup development environment	17/05/2021	7
Setup toolchain to export IML to Ocaml	24/05/2021	7
Develop IML library to represent NNs	31/05/2021	7
Develop a converter to import existing NNs	07/06/2021	7
API to specify properties	14/06/2021	7
Verify properties on NN	21/06/2021	7
IML properties to represent convolution and maxpool operations	28/06/2021	7
Multi-OS compatibility	05/07/2021	7
Evaluation		
Download benchmark datasets	24/05/2021	2
Train models on benchmark datasets	26/05/2021	14
Run evaluations	28/06/2021	7
Run competing methods on same hardware for comparison	05/07/2021	7
Compare results	12/07/2021	3

Figure C.1: A Gantt chart for this project's tasks.