CheckINN: Wide Range Neural Network Verification in Imandra

Remi Desmartin  
Heriot-Watt University  
Edinburgh, UK  
rhd2000@hw.ac.uk

Ekaterina Komendantskaya∗  
Heriot-Watt University  
Edinburgh, UK  
ek19@hw.ac.uk

Grant Passmore  
Imandra Inc.  
Austin, TX, USA  
grant@imandra.ai

Matthew Daggitt†  
Heriot-Watt University  
Edinburgh, UK  
md2006@hw.ac.uk

ABSTRACT
Neural networks are increasingly relied upon as components of complex safety-critical systems such as autonomous vehicles. There is high demand for tools and methods that embed neural network verification in a larger verification cycle. However, neural network verification is difficult due to a wide range of verification properties of interest, each typically only amenable to verification in specialised solvers. In this paper, we show how Imandra, a functional programming language and a theorem prover originally designed for verification, validation and simulation of financial infrastructure can offer a holistic infrastructure for neural network verification. We develop a novel library CheckINN that formalises neural networks in Imandra, and covers different important facets of neural network verification.

CCS CONCEPTS
• Theory of computation → Logic and verification; Higher order logic; Program verification.

KEYWORDS
Neural Networks, Verification, Robustness, Boyer-Moore Provers

ACM Reference Format:
Remi Desmartin, Grant Passmore, Ekaterina Komendantskaya, and Matthew Daggitt. 2022. CheckINN: Wide Range Neural Network Verification in Imandra. In 24th International Symposium on Principles and Practice of Declarative Programming (PPDP 2022), September 20–22, 2022, Tbilisi, Georgia. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3551357.3551372

1 MOTIVATION
Machine learning algorithms have recently become a key technology underlying complex autonomous systems such as autonomous cars, chatbots or intelligent trading agents. Neural network (NN) is an umbrella term for a large family of machine-learning algorithms. Abstractly speaking, a neural network \( F \) is a function of type \( \mathbb{R}^m \rightarrow \mathbb{R}^n \). We usually understand that this function is obtained by fitting the function’s parameters to give an optimal assignment of the available data (given by points in an \( m \)-dimensional space) to \( n \) classes. The process of fitting such a function is usually called training or learning, and the optimisation algorithms used in the process are called learning algorithms.

Because learning algorithms rely on incomplete and often noisy data, the solutions they offer are difficult to verify with standard safety assurance methods. One safety verification scenario is to prove that a neural network will never misclassify “important” inputs. This condition has several mathematical approximations [5]: e.g. draw an \( \epsilon \)-ball around each important data point and prove that all images within those \( \epsilon \)-balls are classified correctly [19]. This style of neural network analysis is often called robustness verification, as we prove a network robust to image change within \( \epsilon \)-perturbation. The verification community has proposed several algorithms for robustness verification, a majority of which are based on either SMT-solving [19, 22] or abstract interpretation [1, 15, 35]. The main limiting factors for robustness verification are poor scalability to large or non-linear neural networks, and the limited scope of \( \epsilon \)-ball robustness as a safety property.

Functional programming (FP) and interactive theorem provers (ITPs) have so far played only a marginal role in the domain of neural network verification. There is a library [30] formalising small rational-valued neural networks in Coq and proving their structural properties. A more sizeable formalisation called MLCert [2] imports neural networks from Python, treats floating point numbers as bit vectors, and proves properties describing the generalisation bounds for the neural networks. An \( F^* \) formalisation [25] uses \( F^* \) reals and refinement types for proving robustness of networks trained in Python. Each approach had its own limitations. For example, MLCert does not prove neural networks’ robustness, the \( F^* \) formalisation only proves robustness; neural networks in [30] are too small for machine learning applications that we seek to verify.

At the same time, one lesson that successful industrial provers like Isandra [31, 32] teach us is that real life verification efforts require a wide range of facilities, such as (a) user-friendly higher-order syntax, (b) ability to execute the code in order to prototype the system’s behaviour and study counterexamples, (c) proof automation for routine proofs, (d) complete techniques for bounded verification (including counterexample synthesis), and (e) facility
The ϵ Neural Networks (CNNs) generalise the standard definition of “fully connected” neural networks (FNNs) by introducing a range of different layer types with different geometry. They are widely used in computer vision as more sophisticated layer geometry allows the network to capture more general features in data. The choice between CNNs and FNNs does not seem to play a crucial role in reachability verification, but this paper shows their potential role in structural verification, as they open new ways of exploring the structural properties of neural networks.

CNNs are challenging for ITP formalisation, as they work with images and expect 2D or 3D input data, and assume that different kinds of “layers” (convolutional, pooling, fully connected) can be composed flexibly to form a neural network, which at the level of formalisation requires a generic approach to layer definition.

CheckINN addresses these technical hurdles.

### Proof Methods

| Verification Property | Proof Method              | Type of NN | Matrix Representation | Numeric choice |
|-----------------------|---------------------------|------------|-----------------------|----------------|
| Sec. 4 Structural     | Induction, Imandra Waterfall | FNN, CNN  | Lists                 | Real, Integer  |
| Sec. 5 Reachability (ε-ball robustness) | SAT-solver Blast | FNN, CNN | Lists                 | Integer        |
| Sec. 6 Reachability (ACAS Xu) | Imandra Waterfall | FNN        | Functions, Records    | Real, Integer  |

**Figure 1:** The range of verification design decisions covered in each section of this paper.

- **Sec. 4 Structural:** Induction, Imandra Waterfall
- **Sec. 5 Reachability (ε-ball robustness):** SAT-solver Blast
- **Sec. 6 Reachability (ACAS Xu):** Imandra Waterfall

Usually, proving such a property requires induction on the structure of F (as well as possibly nested induction on parameters of Q). As such proofs rely on structural properties of F captured in Q, we will call verification properties stated in this form **structural properties**. However, as this paper will show, finding such structural properties is by no means an easy task (unless the networks are small [30]).

This is why the verification community often resorts to proving properties like

\[\mathcal{P}^R: \text{if the given neural network } F, \text{ if a property } R_1 \text{ holds for its inputs, verify that a property } R_2 \text{ holds for } F's \text{ outputs.}\]

The ε-ball robustness [1, 15, 19, 35] or ACAS Xu challenges [22] are formulated in this way. Because this kind of verification proof exploits how a property of inputs R1 propagates through the given neural network, we call verification properties stated in this form **reachability properties**.

The choice of properties determines the choice of proof methods, which we will call respectively **structural proofs** and **reachability proofs**. Crucially, Imandra can perform both structural and reachability proofs, which distinguishes it from neural network solvers like Marabou [22] or ERAN [15, 35]. Indeed Section 6 shows that Imandra uses its original proof strategies in the reachability proofs of the ACAS Xu challenge [22]. However, without any further domain-specific heuristics or proved libraries of lemmas, it does not match the performance of the domain-specific verifiers.

2. **The choice of neural network architecture.** Convolutional Neural Networks (CNNs) generalise the standard definition of “fully connected” neural networks (FNNs) by introducing a range of different layer types with different geometry. They are widely used in computer vision as more sophisticated layer geometry allows the network to capture more general features in data. The choice between CNNs and FNNs does not seem to play a crucial role in reachability verification, but this paper shows their potential role in structural verification, as they open new ways of exploring the structural properties of neural networks.
the style of Boyer-Moore [4]). For the nonlinear case, Imandra supports reasoning with real algebraic numbers [8]. Moreover, as Imandra supports recursion and higher-order functions, non-polynomial real functions may be defined and reasoned about by defining recursive functions which approximate them via, e.g., Cauchy sequences.

**CheckINN** capitalises on Imandra’s real number facilities and this paper makes a point of studying where, and how, transitioning between real-valued and quantised matrices makes a difference for neural network verification.

The paper is structured as follows. Section 2 gives necessary background on neural networks and Imandra syntax. Section 3 introduces an implementation of CNNs in Imandra. Section 4 presents the first verification task – a proof of a structural property (neural network monotonicity) for FNN, and at the same time illustrates Imandra’s waterfall method. It then considers a more difficult scenario of formulating and proving structural properties of CNNs. Section 5 uses Blast, Imandra’s SAT solver on the matrix-as-list version of this paper in [11]).

In Imandra, we aim to define NNs as functions that compose layers:

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

To implement this in FP, we have three key choices:

1. to represent matrices as lists of lists (and take advantage of the inductive data type List),
2. define matrices as functions from indices to matrix elements,
3. or take advantage of record types, and define matrices as records with maps.

In the accompanying note [10] we focus specifically on the technical consequences of taking each of these choices in Imandra. Here, we will build up our matrix representations gradually, explaining the consequences of various choices as we go.

We start with lists (cf. Listing 1). Most list manipulation functions of the OCaml List module are available in Imandra, which opens the way for library re-use. For example, the definition of a dot product uses map2 (a straightforward generalisation of List.map to matrices) and the sum function, which in turn applies List.fold_left and + to vectors. A fully connected layer is then defined as a function fc which takes as parameters an activation function, a 2-dimensional matrix of layer’s weights and an input vector. Note that each row of the weights matrix represents the weights for one of the layer’s nodes. 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.
we present performs explicit dimension checking via a result feature map. The filter shows a horizontal line pattern. The bottom area of the input image. The filter shows a horizontal line pattern. The bottom area of the input image matches the filter better than the top one, resulting in higher values in the feature map.

Figure 3: A feature map resulting from a convolution operation, given an image. The filter shows a horizontal line pattern. The bottom area of the input image matches the filter better than the top one, resulting in higher values in the feature map.

Figure 4: Max pooling operation with a 4 × 4 filter. Each coloured zone is a region where the filter is applied.

It is now easy to see that our desired approach to composing layers given in Listing 2 works as stated. We may define the layers using the syntax: let layer_i = fc a weights, where i stands for 0,1,2,3, and a stands for any chosen activation function.

Although natural, this formalisation of layers and networks suffers from two problems. Firstly, it lacks the matrix dimension checks that were readily provided via refinement types in [25]. This is because Imandra is based on a computational fragment of HOL, and has no refinement or dependent types. To mitigate this, the library we present performs explicit dimension checking via a result monad (indeed the code in Table 1 gives a good idea of dimension error tracking in this part of CheckINN). Secondly, the matrix definition via the list data types makes unrolling-based [32] proofs of robustness inefficient, as even accessing matrix elements typically involves unfolding several layers of recursion. In Section 6 we will present a more efficient approach that defines matrices as functions, and alleviates both of these problems. However, we proceed with this simpler data type definition of matrices for the time being.

3 CNNS IN IMANDRA

We now introduce the CNN part of CheckINN. Unlike their fully-connected counterparts, CNNs are designed to make use of spatial information that may be present in data. To illustrate this, consider the following artificial data set created to serve as a running example for this paper. It consists of 144 unique images of dimension 9 × 9 × 1 (see Fig. 2), in which images are classified as happy or sad faces. This toy data set makes it easier for us to expose the main ideas behind CNN verification. In later sections, we will be using ACAS Xu data set and networks [22] as well.

The best way to recognise a smile or a frown is by considering the spatial configuration of pixels around the mouth. If these pictures were flattened into vectors, the spatial information would be lost. In order to analyse 2D data, layers of different kinds (convolutional, pooling and fully connected) operate over the submatrices of the matrix that represents a given data point, as illustrated in Fig. 5. We first describe how the Imandra library defines each of these layers.

3.1 Convolutional Layer

Convolutional layer weights are given by several multidimensional matrices, called filters. Each filter captures some distinct “feature”. For example, given three filters, each can detect respectively diagonal, vertical and horizontal lines present in an image. The layer output is the result of convolution operations between the input image and each of its filters. For each filter, the layer outputs one 2-dimensional array called a feature map. Fig. 3 shows a convolution operation between an image and a filter.

Definition 3.1 (Convolution Operation). Let J be a 2-dimensional matrix of size $h_j \times w_j$, and let $K$ be a 2-dimensional square filter of size $k \times k$, then the feature map $M$ is the result of the convolution operation between $J$ and $K$, defined as follows. $M$’s dimensions are $(h_j - k + 1) \times (w_j - k + 1)$, and the value of its elements at the intersection of the $i^{th}$ row and $n^{th}$ column is determined by the equation: $M_{i,n} = \sum_{k=1}^{k} \sum_{p=1}^{p} K_{n,p} J_{i+s,n+p}$. To make it more amenable to formalisation in Imandra, we will slightly rewrite this definition. By using $X[i,i+s,n,n+k]$ to denote the submatrix of a matrix $X$ formed by the intersection of the rows $i_1$ to $i_s$ and columns $n_1$ to $n_t$, we use:

$$M_{i,n} = K \cdot J[i,i+k; n,n+k].$$

Because filters are intended to represent features, i.e. patterns characteristic of a class, a convolution operation between filters and an input matrix can be seen as checking which part of the input matches the feature present in the filter; hence the name “feature map” for its result. We formalise the convolutional layer in Listing 3. The function convolution implements a convolution operation between a matrix and a filter; fold_left iterates the operation.

So far, we assumed that the input matrix only has one colour channel, but images usually have three. To apply a convolution operation to input with multiple channels, the filters must have the same number of channels. CheckINN handles such cases.

3.2 Pooling Layer

Pooling layers come after convolutional layers; their input is the feature maps from the previous layer, and their output is a set of
let rec convolution_row' input filter (row, col) =  
let (row', col') = Matrix.dimensions filter in  
if col < 0 then Ok [] else  
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  (* col decreases to let imandra prove termination *)  
Res.bind2 head dot_p (fun x y -> Ok (x @ [y]))

let convolution_row input filter row =  
let (i_rows, i_cols) = Matrix.dimensions input in  
let (f_rows, f_cols) = Matrix.dimensions filter in  
if i_rows < f_rows then Error " convolution_row : filter's height is greater than input's" else  
if i_cols < f_cols then Error " convolution_row : filter's width is greater than input's" else  
let col = (i_cols - f_cols) in  
convolution_row' input filter (row, col)

let convolution (input : real Matrix.t) (filter : real Matrix.t) =  
if not (Matrix.is_valid input) || not (Matrix.is_valid filter) then Error " convolution : invalid matrix" else  
let (i_rows, _) = Matrix.dimensions input in  
let (f_rows, _) = Matrix.dimensions filter in  
if i_rows < f_rows then Error " convolution : filter's size is greater than input's" else  
let acc_fun (_, xs) _ =  
  if f_rows + i > i_rows then (i + 1, xs) else  
  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 3: A representative snapshot of CheckINN code for CNN.

Figure 5: Representation of a CNN’s layers and intermediate outputs with their dimensions.

2-dimensional matrices that reduce feature maps in size. Two main types of pooling operations are used in CNNs: max pooling and average pooling. By abuse of terminology, the literature also refers to filters in the pooling layer (cf. Fig 4), but in fact “filters” here simply define the submatrix size. Our formal definition clarifies this point.

Definition 3.2 (Max Pooling Operation). Given a 2-dimensional input matrix \( J \) and the filter of size \( k \), the max pooling operation is a function that returns a matrix \( M \), whose elements are defined by \( M_{i,n} = \max [J[i, i + k; n, n + k]] \).

The CheckINN implementation of pooling layers closely mimics the style of formalisation of the convolutional layer, except for using the max operation instead of the dot product. Average pooling layers work the same way, but instead of a max function, they use an averaging function.

3.3 Assembling All Layers

CNNs alternate between convolutional layers and pooling layers. This allows them to achieve a “greater level of abstraction” in feature detection. A typical CNN architecture usually chains several convolutional layers and pooling layers. The flattening layer flattens the 3D representation into a vector, and several fully connected layers complete the network.

We can now assemble the network shown in Fig. 5, by using the syntax of Listing 2. To do this, we need to import a trained neural network from Python. CheckINN uses Keras to train our networks, and it contains a Python script to convert a CNN saved in Keras [6] format into an Imandra module containing each layer’s weights. The layers must then be instantiated with layer functions. The Layers module encapsulates the individual layer modules to expose higher-order functions that instantiate layer functions. The convolution and max pooling layers are implemented in their respective modules for a single filter but Layers can hold several filters; all filters are then flattened together. These functions are partially applied to a multi-dimensional array of weights, to create layer functions that can be chained to form a network:

```ocaml
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 model input = layer_0 input >>= layer_1 >>= layer_2 >>= layer_3
```
As these layer functions are implemented in a generic 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 expected input. (The dimensions are checked dynamically, and we will see errors at run time if the dimensions do not match.) For instance, an FNN can be created by chaining only fully connected layers. Note that the dimensions of layer inputs and outputs are not specified in the user interface to the library, they are deduced from the layer dimensions.

4 STRUCTURAL PROPERTIES

When proving structural properties of neural networks, we are interested in showing how a certain feature present in a network’s architecture influences its behaviour as a function. As a consequence, such proofs quantify over all neural networks with said architectural features; and usually require induction on the network’s structure. This style of proof matches best with Imandra’s original design, as a higher-order inductive theorem prover.

We start with a simple example of a monotonicity property for a FNN and use it to also illustrate Imandra’s Waterfall proof method. We then investigate structural properties of CNNs that may be useful in verification.

4.1 Monotonicity and Inductive Proofs

There has been some interest in monotone networks in the literature [34, 38]. We will emulate a monotonicity property as follows: any fully connected network with positive weights is monotone, in the sense that, given increasing positive inputs, its outputs will also increase.

For the sake of this section, we somewhat simplify the code for FNNs. Taking lists of real numbers for input $i$, 2D and 3D matrices (as lists) for the weights ($w$) and biases ($b$), we define:

```coq
let rec layer ws bs i = match (ws, bs) with
| (_, []) | ([], _) -> []
| (w::ws, b::bs) -> (perceptron w b i) :: (layer ws bs i)

let rec network ws bs i = match (ws, bs) with
| (_, []) | ([], _) -> i
| (w::ws, b::bs) -> network ws bs (layer w b i)
```

The monotonicity theorem is then stated simply as:

```coq
theorem network_monotonicity ws bs i i' = positiv_3d ws && positiv_2d bs && positiv i && gte i' i
=>> gte (network ws bs i') (network ws bs i)
```

where the positivity conditions at the top are for vectors, matrices and 3D matrices respectively; and $gte$ stands for “greater or equal” applied pointwise to list elements. Note that the theorem above quantifies over FNNs of any size.

Imandra’s proofs are based on the Boyer-Moore waterfall [3] strategy, which automates induction, by generating plausible induction principles, and also applying several heuristics to discharge intermediate goals. The waterfall (@[auto]) proceeds in five steps:

1. **simplification** makes use of all enabled rewrite and forwarding-chaining rules, decision procedures for algebraic data types and arithmetic, and case-splits,

2. **definition unrolling** searches for counterexamples up to a certain unrolling depth (not unlike say in QuickCheck [7]).

3. **destructor elimination** transforms all expressions of inductive types from a destructor form (e.g. $a = \text{List.hd } x$ and $b = \text{List.tl } x$) into a constructor form (e.g. $x = a:b$).

4. **fertilisation** performs rewriting on terms in the goal using equivalent terms defined in the lemma assumptions,

5. **generalisation** generalises the given conjecture.

Then Imandra generates an induction scheme for the generalised goal and restarts the search for a proof from item 1. Although much of this process is automated, Imandra switches to an interactive mode when the proof search fails and suggests missing lemmas.

Let us see Imandra’s waterfall in action (see also [11] for full user dialogue). The proof of monotonicity does not succeed immediately by [@[auto]]. On the first attempt, Imandra realises it has to use induction, and finds six possible ways to proceed by induction. It however manages to simplify the six to two, and finds a clear winner among the two, with induction on the structure of $ws$ and $bs$. To finish the proof, we prove two lemmas (derived by analysing Imandra’s “simplification checkpoints” for the goal):

```coq
lemma positive_push_2d bs ws i =
positive bs && positive_2d ws && positive i
=>> positive (layer ws bs i) [@[auto] @[rw]]

lemma layer_monotonicity ws bs i i' =
positive_2d ws && positive bs && positive i && gte i' i
=>> gte (layer ws bs i') (layer ws bs i) [@[auto] @[rw]]
```

The first lemma shows that positivity is inherited from layer inputs to layer outputs; and the second asserts the monotonicity property for individual layers. Then Imandra completes the inductive proof by @[auto]. Automation of inductive proofs is a strength of Imandra.

Similar Coq proofs in [30] required more tactic guidance.

4.2 Structural Properties of CNNs

It may seem that general structural properties like monotonicity are not very useful for practical verification tasks, especially in CNN verification. In this section, we present an example that shows a structural approach to CNN verification and exposes how general mathematical properties like monotonicity can play a role in the process. We continue to work with the same toy image data set and the CNN $F$ (of Fig. 5). But we highlight the general pattern in the approach that can be extrapolated to other CNNs.

1. Filter Adequacy. We assume that filters bear some structural meaning. For example, a set of CNN filters that would agree with the human definition of a smile could give rise to heatmaps that show diagonal lines, as on the left of Fig. 6. These diagonals are then detected in the later layers of the CNNs, as Fig. 7 shows.\footnote{Recall that filters are real matrices; heatmaps are visualisations of such matrices, where intensity of colours corresponds to values of matrix elements. In particular, in Fig. 6, black stands for 0 and white for 1.}

![Figure 6: Heatmaps of $2 \times 2 \times 1$ filters: human-imposed on the left; learnt by the CNN on the right.](image-url)
In reality, the situation is a bit more complicated. When we train a 100% accurate CNN on this data set and examine heatmaps of the filters it learnt, we notice that in fact it learns the filters shown on the right of Fig. 6. It is not immediately clear how to interpret what the CNN thinks a smile is. However, digging deeper into the layers, we found that both manually constructed and learnt filters give rise to a well-defined pattern in the pooling layer, so it seems that analysis of the pooling layer is crucial. We thus want to define a filter to be adequate for recognition of a smile if it gives rise to a well-defined pattern in the pooling layer, giving a smiling face as an input. The pattern is given by two small diagonals in the bottom corners of the matrix:

\[
\text{let has_pattern } (m: (('a Matrix.t) Vec.t)) : (bool, 'b) = \\
\text{result } = \text{Ok (max_bottom_left_corner (Vec.nth } 0 \text{ m) \&\& } \\
\text{max_bottom_right_corner (Vec.nth } 1 \text{ m))}
\]

2. Definition of Verification Property. Next, we assume that a definition of a smiling face is a specification that can be written by a human. Such a specification usually captures idealised structure and abstracts away from any exceptions. For example, for the given data set, CheckINN defines a happy face as the one that smiles, and a smile as a shape with a left diagonal and a right diagonal on either side of the mouth region, connected by a horizontal line.

The need for neural networks arises when we want to implement systems that deal with noise and exceptions. For example, the picture may be distorted or taken from a wrong angle. In such cases, an idealised rule would fail, whereas a neural network may still succeed. In our verification scenario, we want to ensure some form of soundness, i.e. prove that all cases that fall under the human specification of a happy face are classified as happy: given a CNN \( F \) with adequate filters and a well-tuned fully-connected layer, any image that satisfies the specification of a happy face will always be classified by \( F \) as happy.

The theorem misses the definition of a well-tuned fully-connected layer. From the engineering point of view, the weights of that layer must be tuned to higher values exactly where the “hot” pattern in the pooling layer of an adequate filter is expected. And this is the point that requires a more general mathematical reasoning.

3. Extreme Values Lemma. To understand the problem recall the role of the fully connected layer in a CNN. Let \( x \) and \( y \) denote the two neurons of the output layer \( out \), standing for classes “Happy” and “Sad”. Each of these neurons represents the score for a class for a given input image. The weights associated with \( x \) and \( y \) are respectively denoted by \( w^x = (w^x_1, w^x_2, \ldots, w^x_n) \) and \( w^y = (w^y_1, w^y_2, \ldots, w^y_n) \). After a certain pattern in \( a \) (happy) image is detected in the pooling layer, the pooling layer is flattened into a vector of weights which are used to compute the vector that the units \( x \) and \( y \) receive. So, ultimately, it is now up to neurons \( x \) and \( y \) to classify the pooling layer pattern into one of the two classes.

Let \( a \) denote the input vector of the layer \( out \) and \( a^* \) – the mean value of \( a \). Without loss of generality, let us ignore the activation function of the layer \( out \), and just concentrate on its dot products. For unit \( x \), it calculates \( (a_1 w^x_1 + \ldots + a_n w^x_n) \) and for unit \( y \) it calculates \( (a_1 w^y_1 + \ldots + a_n w^y_n) \). It then decides on the class of the CNN input based on checking

\[
(a_1 w^x_1 + \ldots + a_n w^x_n) > (a_1 w^y_1 + \ldots + a_n w^y_n)
\]

We are almost led to believe that we simply need some monotonicity property that shows that \( w^x > w^y \). However, this property would not apply to CNNs we find in practice. In reality, each of the weight vectors \( w^x \) and \( w^y \) has higher values for certain indices (and lower for others) depending on which features are characteristic of which class. Intuitively, if we know that a smile means high values in the bottom corners of the pooling layer, then it is specifically for these regions that \( w^x \) will have higher values than \( w^y \). So, the general property that we need should describe how well the fully-connected layer is tuned to these “extreme values”.

Let \( a_{\text{max}} \) and \( a_{\text{min}} \) be the max and min values in \( a \). We define extreme values of \( a \) as

\[
a_{\text{ex}} = \{ a_i : a_i > a^* + \frac{a_{\text{max}} - a^*}{2} \}
\]

We say a vector \( a \) has a distinct pattern if all values of \( a \) are either extreme or below \( a^* \).

Lemma 4.1 (Extreme Values). Let \( a \) be a vector \( a_1, \ldots, a_n \) with distinct pattern and extreme values \( a_1, \ldots, a_m \). If for \( w^x \) and \( w^y \), we have

\[
w^x_i > w^y_i \wedge \ldots \wedge w^x_m > w^y_m,
\]

then

\[
(a_1 w^x_1 + \ldots + a_n w^x_n) > (a_1 w^y_1 + \ldots + a_n w^y_n).
\]

Just as in the case of monotonicity, we note that the lemma is stated in full generality, and does not depend on a specific network architecture or a data set. Without further constraining the vectors \( a, w^x \) and \( w^y \), the lemma does not hold. Defining necessary restrictions on these vectors ultimately gives possible definitions of the “well-tuned fully-connected layer”. Indeed, Imandra’s facility for counter-example generation may serve as an aide in formulating the new conditions.

In particular, the lemma holds for the following two special cases (see the proofs in [11] or in CheckINN):

- **R1.** \( a \) is a binary vector, and \( a_{\text{min}} \neq a^*, a_{\text{max}} \neq a^* \);
- **R2.** \( a \) has positive values, \( w^x \) and \( w^y \) are binary vectors, and \( m \geq 1.5 \frac{n}{a_{\text{max}}} \).

Manual proofs of these two cases are short (see [11]), but assume some facts about the relations between \( a^*, a_{\text{max}}, a_{\text{mean}}, a_{\text{ex}} \), which would be laborious to formalise. Our methodological interest here is to show that Imandra can ease one’s verification tasks, rather than complicate them. For **R1**, we can formalise the lemma in a way that will guide Imandra’s inductive proof in the right direction. In particular, we can incorporate our knowledge of extreme values into definition of the dot product:
Listing 4 shows the full interaction with Imandra for proving
ground SMT problem which is amenable to
decision procedures. Completely unrolled and eliminated by Imandra,
and what is left is an explicitly given list structure, all recursive functions can be
for example, unbounded reals or integers. The key point is that with a fixed
results over bounded structures, even if these structures contain,
another way it can be used in proofs: we can prove
symbolic bounded model checking modulo
ground decision procedures, there is another way it can be used in proofs: we can prove
straightforward albeit lengthy, and can be found in [11]. However, this time Imandra
cannot complete the proof automatically. Once again, we will try to avoid burdensome auxiliary lemmas, and
instead showcase yet another useful Imandra tactic: [@unroll]
We already mentioned that unrolling in Imandra plays a role of a
counterexample finder. But, since it is based on the idea of
symbolic bounded model checking modulo ground decision procedures, there is another way it can be used in proofs: we can prove
results over bounded structures, even if these structures contain,
e.g., unbounded reals or integers. The key point is that with a fixed
explicitly given list structure, all recursive functions can be completely
unrolled and eliminated by Imandra, and what is left is a ground SMT problem which is amenable to decision procedures.
Listing 4 shows the full interaction with Imandra for proving R2 for
vectors of dimension 8, which is the maximal dimension we were able to verify with a 300-second timeout. This form of bounded verification is very useful for analyzing concrete conjectures, and may suffice for many verification scenarios in which the architecture of networks is known in advance.

From the methodological point of view, conditions like R1 and R2 give us a way to construct CNNs that can in principle be proved sound. For example, to obtain CNNs that satisfy R1, we would need to apply a binary threshold function on activations of the pooling layer. To obtain R2, we would need to use algorithms that binarise the weights when training. Having these conditions, assembling the other components of the soundness theorem is trivial.

It is out of the scope of this paper to seek more liberal restrictions to the Extreme Values Lemma; however, this would be the future line of work for any scalable project on structural verification of CNNs that follows the described verification scenario. The restrictions above suggest that the key to extending the lemma’s applicability is to find more sophisticated formulae that characterise the relationship between the magnitude and the number of extreme values.

5 REACHABILITY AND SYMBOLIC EXECUTION

We will now apply CheckINN to reachability verification problems. The most popular reachability property in neural network verification is robustness. Informally, a CNN’s robustness is its ability to correctly classify an input to which a small perturbation is applied. More specifically, a CNN is \( \epsilon \)-ball robust for an image if, whenever the distance between the perturbed image and the original is no more than \( \epsilon \), the CNN classifies the perturbed image correctly.

Different techniques exist to ensure network robustness during training: data augmentation [33], adversarial training [29], or training with logical constraints [14]; and [5] shows that these different methods give rise to different formal definitions of robustness, which we summarise in Fig. 8. All properties can be written in first-order logic, and in general are amenable to SMT solvers. Imandra can also express these properties, with the benefit of a somewhat more intuitive syntax than the solvers admit. For example, this is CheckINN definition of standard robustness (using \( L_0 \)-norm distance function on vectors):

We refer the reader to CheckINN code for the remaining three robustness definitions, which use similar syntax. We note the addition of a parameter constraint on admissible CNN inputs, which we
often use as a validity check for the type of input images that the network accepts, as will be illustrated later in this section.

Robustness is best amenable to proofs by arithmetic manipulation. This explains the interest of the SMT-solving community in the topic, which started with using Z3 directly [19], and has resulted in highly efficient SMT solvers specialised in robustness proofs for neural networks [22, 24].

In Imandra, [0@blast], a tactic for SAT-based symbolic execution modulo higher-order recursive functions, can be applied to these problems. However, blast currently does not support real arithmetic. This requires us to quantise the neural networks we use (i.e. convert them to integer weights) and results in a quantised CNN library in CheckNN. Quantisation is a common technique in machine learning and NN verification: quantised neural networks take less computational resources to run, are more amenable to verification, and often can be trained to be as accurate as floating point networks [12, 26, 27]. Modulo this hurdle, verification of the CNNs goes in a straightforward way, and requires just one line of code. For example, for standard robustness, this line looks like this:

Note that the code includes the validity check for input images; for example, we may require that all input matrices are of size $9 \times 9$ and have binary inputs. This reduces the search space and gives more tractable results. This is also the first instance when we use the tactic syntax [0@blast]. Imandra’s mode of interaction is by supplying proof details and hints to the user, and taking additional lemmas and tactics like [0@blast] as input. In this case, just calling [0@blast] completes the proof.

To illustrate the usual pattern of robustness verification, we select images from the data set, for example those shown in Fig. 2 and use the module that holds all verification calls as in the code above. We obtain the results shown in Fig. 9. We can see that all the properties terminated and Imandra gives a “proved” or “refuted” result. In the latter case, Imandra generates an executable counterexample which is a benefit of the language.

The execution times given in Fig. 9 are reasonable, but the example network and images are rather small. Already a quantised version of the ACAS Xu challenge [22] is out of reach for [0@blast], which we will try to repair in the next section.

Our conclusions are two-fold. Firstly, we notice the payoff of implementing a large general library for CNNs: we can now implement and verify robustness properties in just a few lines of code, in a clear syntax. This shows Imandra’s ease of use as a verification tool. Secondly, we managed to experimentally confirm the suggestion by [5] 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. And finally, this points to a future research direction – connecting Imandra with neural network-specific solvers like Marabou, in which case a call of a procedure similar to [0@blast] could perhaps deal with the queries more efficiently; moreover, it would open the way for such proofs in the real-valued version of our CNN library.
Similarly to robustness properties, this verification property could be handled by general-purpose SMT solvers; however, as [22] points out, they do not scale. Indeed, when we use the CheckINN on quantised ACAS Xu neural networks, [28] last does not terminate. This is why the algorithm Reluplex was introduced in [22] as an additional heuristic to SMT solver algorithms; Reluplex has since given rise to a domain specific solver Marabou [24].

6.1 Leveraging Imandra’s Native Automation: Matrices As Functions

We start with keeping the integer values for weights but redefining matrices as functions (from indices to values), which gives constant-time (recursion-free) access to matrix elements:

```ocaml
type arg =
  | Rows
  | Cols
  | Value of int * int

type 'a t = arg -> 'a

let nth (m: 'a t) (i: int) (j: int): 'a = m (Value (i, j))
```

Note the use of the `arg` type, which treats a matrix as a function evaluating "queries" (e.g., "how many rows does this matrix have?" or "what is the value at index (i, j)?"). This formalisation technique is used as Imandra’s logic does not allow function values inside of algebraic data types. We thus recover some functionality given by refinement types in [25].

Furthermore, we can map over a matrix, map2 over a pair of matrices, transpose a matrix, construct a diagonal matrix etc. without any recursion, since we work point-wise on the elements. At the same time, we remove the need for error tracking to ensure matrices are of the correct size: because our matrices are total functions, they are defined everywhere (even outside of their stated dimensions), and we can make the convention that all matrices we build are valid and sparse by construction (with default 0 outside of their dimension bounds).

For full definitions of matrix operations and layers, the reader is referred to CheckINN, but we will give some definitions here, mainly to convey the general style (and simplicity?) of the code. A script transforms the original ACAS Xu networks into a sparse functional matrix representation. For example, layer 5 of one of the networks we used is defined as follows (fc stands for a fully-connected layer):

```ocaml
let layer5 = fc relu {
  function
  | Rows -> 50
  | Cols -> 51
  | Value (i,j) -> Map.get (i,j) layer5_map
}
```

Networks are compressed in order to reduce the number of computations using two well-known compression methods. On one hand, they are quantised, i.e. the real-valued weights are converted into integers using static quantisation [27]. On the other hand, the weights are pruned using magnitude as a pruning criterion, meaning that weights with the lowest absolute value are removed.

We can model the resulting neural network via a function run:

```ocaml
let run (dist, angle, angle_int, vown, vint) =
  let m = mk_input (dist, angle, angle_int, vown, vint) in
  layer0 m |> layer1 |> layer2 |> layer3 |> layer4 |> layer5 |> layer6
```

Note that we no longer need to use monadic binds, as we no longer track dimension errors. We can now define the first ACAS Xu property [22]:

```ocaml
let property1 x =
  is_valid x && condition1 x => property1 x
```

The only help Imandra needs to prove this automatically are the forward-chaining rules about the relu function:

```ocaml
lemma relu_pos x =
  x >= 0 => (relu x) [trigger] = x
[@auto] [00fc]

lemma relu_neg x =
  x <= 0 => (relu x) [trigger] = 0
[@auto] [00fc]
```

And then we disable relu expansion for all of the proofs using the [00disable] annotation. This way, relu induces no simplification case-splits, while all relevant information about relu values is propagated, per instance, on demand to our simplification context. Now Imandra’s engine takes care of the proof automatically (when we use the tactic [@auto]), and takes just under 1.5 minutes. In Figure 10 we give a representative evaluation of Imandra’s performance on several ACAS Xu networks and properties. We set the timeout time to 5 minutes, and approximately half of the cases terminate within the time limit. Execution time is orders of magnitude faster than Marabou’s time on full ACAS Xu networks, which may suggest that combining pruning and verification [28] is a good direction for Imandra. We leave a thorough investigation of this for future work.

Several factors played a role in automatizing the proofs. Firstly, Imandra being a higher-order functional language opened the way for us to experiment with alternative matrix representations in the first place. By using maps for the large matrices, we eliminate all recursion (and large case-splits) except for matrix folds (which now come in only via the dot product), which allowed Imandra to expand the recursive matrix computations "on demand." Finally, Imandra’s native simplifier contributed to the success. It works on a DAG representation of terms and speculatively expands instances of recursive functions, only as they are (heuristically seen to be) needed. Incremental congruence closure and simplex data structures are shared across DAG nodes, and symbolic execution results are memoized. Moreover, forward-chaining rules (such as those characterising relu) are only applied on demand. Informally speaking, Imandra works lazily expanding out the linear algebra as it is
needed, and eagerly with sharing information over the DAG. Contrast this approach with that of reluplex which, informally, starts with the linear algebra fully expanded, and then works to derive laziness and sharing.

### 6.2 Extension to Reals

Section 2 defined matrices as lists of lists; and that definition in principle worked for both integer and real-valued matrices. However, we could not use [@@blast] to automate proofs when real values were involved; this meant we were restricted to verifying integer-valued networks. The matrix-as-function implementation can be extended to proofs with real-valued matrices; however, it is not a trivial extension. In Section 6.1, the matrix’s value was of the same type as its dimensions. Thus, if the matrix elements are real-valued, then in this representation the matrix dimensions will be real-valued as well. But this complicates termination guarantees for functions which do recursion along matrix dimensions.

To simplify the code and the proofs, three potential solutions were considered:

1. Use an algebraic data type for results of matrix queries: this introduces pattern matching in the implementation of matrix operations, which reduces proof search efficiency.
2. Define a matrix type with real-valued dimensions and values: this poses the problem of proving the function termination when using matrix dimensions in recursion termination conditions.
3. Use records to provide polymorphism and allow matrices to use integer dimensions and real values.

In an accompanying note [10], we provide further details on each of the three implementations in CheckINN. But the second option was a clear winner when it came to evaluating it on ACAS Xu. We therefore only highlight its features here. The implementation is symmetric to the one using integers:

```plaintext
type arg =
| Rows
| Cols
| Value of real * real

type 'a t = arg -> 'a
```

A problem arises in recursive functions where matrix dimensions are used as decrementors in stopping conditions, for instance in the fold_rec function used in the implementation of the folding operation. Imandra only accepts definitions of functions for which it can prove termination. The dimensions being real numbers prevents Imandra from being able to prove termination without providing a custom measure. In order to define this measure, we need to connect the continuous world of reals with the discrete world of integers (and ultimately ordinals) for which we have induction principles. We chose to develop a floor function that allows Imandra to prove termination with reals.

To prove termination of our fold_rec function recursing along reals, we define an int_of_real : real -> int function in Imandra, using a subsidiary floor : real -> int which computes an integer floor of a real by “counting up” using its integer argument. In fact, as matrices have non-negative dimensions, it suffices to only consider this conversion for non-negative reals, and we formalise only this. We then have to prove some subsidiary lemmas about the arithmetic of real-to-integer conversion, such as:

```plaintext
lemma floor_mono x y b =
  Real.(x <= y && x >= 0. && y >= 0.)
  ==> floor x b <= floor y b

lemma inc_by_one_bigger_conv x =
  Real.(x >= 0. ==> int_of_real (x + 1.0) > int_of_real x)
```

Armed with these results, we can then prove termination of fold_rec and admit it into Imandra’s logic via the ordinal pair measure below:

```plaintext
[@@measure Ordinal.pair
  (Ordinal.of_int (int_of_real i))
  (Ordinal.of_int (int_of_real j))]
```

Extending the functional matrix implementation to reals was not trivial, but it did have a real payoff. Using this representation, we were able to verify real-valued versions of the pruned ACAS Xu networks! In both cases of integer and real-valued matrices, we pruned the networks to 10% of their original size. So, we still do not scale to the full ACAS Xu challenge. However, the positive news is that the real-valued version of the proofs uses the same waterfall proof tactic of Imandra, and requires no extra effort from the programmer to complete the proof. Moreover, as preliminary evaluation in Figure 10 shows, the real values do not substantially increase verification times. This result is significant bearing in mind that many functional and higher-order theorem provers are known to have significant drawbacks when switching to real numbers.

### 7 Conclusions, Related and Future Work

#### 7.1 Paper Summary

CheckINN, defined, as broadly as possible, the design space for neural network verification in ITP. As far as we know, no other single existing tool [1, 15, 19, 24, 35] or library [2, 30, 37] has yet managed to cover such a wide range of verification tasks. We have taken advantage of both the wide range of choices for matrix representation available in Imandra when it came to reachability proofs, and the facility to combine first-order and higher-order object definitions, proofs by induction, simplification and decision procedures in the structural proofs.

The resulting CheckINN library contains several parts:

a) a set of modules that use a monadic implementation of matrices as lists that includes a CNN and an FNN implementation, both of which support quantised and real-valued networks (this library was used in Sections 2, 3 and 5);

b) a library with several alternative abstract definitions of FNNs specifically tailored for reasoning by induction (used in Section 4);

c) a library for FNNs using matrices implemented as functions and records for both real- and integer-valued networks (used in Section 6).

They are not currently connected, but complement each other as follows. Part a) is best suited to implement and run CNNs that are compiled directly from a Python environment; and it works reasonably well for robustness verification for small quantised networks. Part b) is built to yield automated inductive proofs that exploit structural properties of FNNs. Part c) performs particularly well with reachability analysis for larger FNNs and supports real numbers.
Figure 10: Results of experiments ran on the properties and networks from the ACAS Xu benchmark [23]. The verifications were run on virtual machines with four 2.6 GHz Intel Ice Lake virtual processors and 16GB RAM. Timeout was set at 5 hours.

| Property | Result | CheckINN: Pruned and Quantised Networks | CheckINN: Pruned Networks | Reluplex: Full ACAS Xu Networks |
|----------|--------|----------------------------------------|---------------------------|--------------------------------|
|          | Quantity | Time (s) | Quantity | Time (s) | Quantity | Time (s) |
| \(\phi_1\) SAT | 4 | 258 | 20 | 13387 | 0 | 394517 |
| UNSAT | 0 | 0 | 0 | 41 | 4 |
| TIMEOUT | 5 | 24 | 26 | 82419 | 4 |
| \(\phi_2\) SAT | 7 | 2098 | 35 | 463 |
| UNSAT | 0 | 2 | 1 | 1 |
| TIMEOUT | 0 | 4 | |
| \(\phi_3\) SAT | 0 | 10453 | 35 | 82419 | 4 |
| UNSAT | 0 | 1 | 1 |
| TIMEOUT | 0 | 4 |
| \(\phi_4\) SAT | 16 | 1422 | 36 | 21533 | 0 |
| UNSAT | 1 | 114 | 0 | 12475 | 0 |
| TIMEOUT | 18 | 3 | 5 | |
| \(\phi_5\) SAT | 1 | 57 | 1 | 98 | 0 |
| UNSAT | 0 | 0 | 0 | 19355 | |
| \(\phi_6\) SAT | 1 | 196 | 1 | 98 | 0 |
| UNSAT | 0 | 0 | 0 | 180288 | |
| \(\phi_7\) TIMEOUT | 1 | 1 | 1 | 40102 | |
| \(\phi_8\) SAT | 1 | 1 | 1 | 0 |
| TIMEOUT | 1 | |
| \(\phi_9\) SAT | 1 | 66 | 1 | 109 | 0 |
| UNSAT | 0 | 0 | 1 | 99634 | |
| \(\phi_{10}\) SAT | 1 | 116 | 0 | 0 | 0 |
| UNSAT | 0 | 1 | 637 | 1 | 19944 |
| TIMEOUT | 0 | 0 | |

7.2 Contributions with Respect to Related Work

CNN formalisation and formulation of structural properties of CNN are both original contributions. We are not aware of any prior similar results in any ITP.

Matrix Representations. We showed that the choice of matrix representation favours certain kinds of proofs. Matrices as lists are well-amenable to structural proofs by induction, while matrices as functions or records help to scale reachability proofs. Flexibility with matrix choices proved to be a useful feature. Real numbers in Imandra allowed for smooth transitions from integer to real parts of the library, especially in inductive proofs.

FP literature gives a selection of different matrix representation methods. Matrices as lists are considered in [18] (in the context of dependent types in Coq), in [25] (in the context of refinement types of F*) and in [17] (for sparse matrix encodings in Haskell). The difference between the list and function approaches was discussed in [39] (in Agda, but with no neural network application in mind). Our main contribution here is to trace the connection between matrix representations and the automation of different kinds of proofs. We believe that the methods we described could be useful in other theorem provers (both first- and higher-order) that combine functional programming and automated proof methods, such as ACL2, PVS, Isabelle/HOL and Coq. For example, in all these systems functions defining matrix operations (e.g., convolution) over lists are often more complex compared to their counterparts over matrices represented as functions, which can benefit from non-recursive definitions. Overall, as these various prominent theorem proving systems work ultimately with functional programs over algebraic datatypes, our core observations carry over to them in a natural way.

Structural Verification of NN. De Maria et al. [30] formalise in Coq "neuronal archetypes" for biological neurons. Each archetype is a specialised kind of perceptron (a small, typically single-unit, neural network), in which additional functions are added to amplify or inhibit the perceptron’s outputs. The paper collects a rich variety of structural properties and proofs characterising these archetypes, formalised in Coq. In this paper, we worked with neural networks used in classification, and unlike [30] had to work with matrices. Defining structural properties for such networks was more challenging. While the monotonicity proofs of Section 4 do not differ much in their complexity from [30] (we may only note the greater power of inductive proof automation that Imandra offers), the proofs of Extreme Values Lemma would have been hard to replicate in Coq as simply as we did it with Imandra. In particular, the bounded model checking tactic used in that section is unique to Imandra.
Reachability Verification of NN. Although Imandra’s simplifier-based automation did not scale to the original dense ACAS Xu network verified by Reluplex [23], we are encouraged that the obtained proofs were achieved without tuning Imandra’s generic proof automation strategies. No other ITP we know of would be able to achieve that much using its native tactics. We are hopeful that the development of neural-network specific tactics will help Imandra scale to bigger networks in the future. Indeed, directly connecting Imandra with Marabou or other similar solvers is also a possible future direction.

7.3 Future Work

Some other considerations were left for future work. This paper draws a wide range of NN verification methods, without aiming at any single verification challenge in particular. Our next step is to apply these methods to some significant verification task. In this respect, extending structural verification of CNN to real-life data sets and scenarios, and further automation of reachability proofs have high priority.

Current state of the art reachability solvers like Marabou [24] rely on SMT solvers, which cannot handle exponents and logarithms, and this explains why they only work with networks with linear activation functions. Although CheckINN does not extend state of the art in this aspect yet, we hope to apply Imandra’s existing experience with using approximating recursive functions (e.g., Cauchy sequences) to non-linear activation functions commonly used in NNs.

There is an important question of the numerical types used in neural networks, that still awaits a successful resolution by the theory proving communities. We used real and integer-valued networks. Mainstream work in Python works with floats, most SMT-based solvers use rationals or restricted reals [22, 24], abstract interpretation tools can use floating points [35] and some ITPs are amenable to formalisations with reals [25] and even floating points [2]. But it is known that transition from one to another may render sound proofs unsound [20], and so the choices cannot be taken lightly.

This paper only addresses the problem of analysing and verifying already trained neural networks. There may be demand for verification of machine learning algorithms (as was done e.g. in [37] for decision stumps), which is worth exploring in the future.

Finally, recent developments in neural network verification suggest that training should become part of verification techniques [5, 14, 36]. Exploring Imandra’s facilities to sustain the “training-for-verification” cycle is left for future work.

REFERENCES

[1] Edward W. Ayers, Francisco Eiras, Majd Hawasly, and Ian Whiteside. 2020. PaToR: A Practical Framework for Robust Deep Neural Network Training. In NASA Formal Methods - 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11-15, 2020, Proceedings (LNCS, Vol. 12289), Springer, 63–84.

[2] A. Bagnall and G. Stewart. 2019. Certifying True Error: Machine Learning in Coq with Verified Generalisation Guarantees. AAK (2019).

[3] R. Boyer and J. Moore. 1979. A Computational Logic. ACM Monograph Series. Academic Press, New York.

[4] Robert S. Boyer and J Strother Moore. 1988. Integrating decision procedures into heuristic theorem provers: a case study of linear arithmetic. Machine intelligence (1988), 83–124.

[5] Marco Casadio, Ekaterina Komendantskaya, Matthew L. Daggitt, Wen Kokke, Guy Katz, Guy Amz, and Idan Belaseli. 2022. Neural Network Robustness as a Verification Property: A Principled Case Study. In Computer Aided Verification (CAV 2022) (Lecture Notes in Computer Science). Springer.

[6] François Chollet et al. 2015. Keras. https://keras.io.

[7] J. Korevaar and John Huisken. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP ’00). Association for Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

[8] Leonardo de Moura and Grant Olney Passmore. 2013. Computation in Real Closed Infinitesimal and Transcendental Extensions of the Rationals. In CADE.

[9] Remi Desmartin, Grant Passmore, Ekaterina Komendantskaya, and Matthew L. Daggitt. 2022. CNN Library in Imandra. https://github.com/aisec-private/ImandraNN.

[10] Remi Desmartin, Grant Passmore, and Ekaterina Komendantskaya. 2022. Neural Networks in Imandra: Mark Representation as a Verification Choice. https://arxiv.org/abs/2205.09556.

[11] Remi Desmartin, Grant Passmore, Ekaterina Komendantskaya, and Matthew Daggitt. 2022. CheckINN: Wide Range Neural Network Verification in Imandra (Extended). https://doi.org/10.48550/arXiv.2207.10562 arXiv:2207.10562 [cs].

[12] Kirsty Duncan, Ekaterina Komendantskaya, Robert J. Stewart, and Michael A. Lones. 2020. Relative Robustness of Quantized Neural Networks Against Adversarial Attacks. In 2020 International Joint Conference on Neural Networks, IJCNN 2020, Glasgow, United Kingdom, July 19-24, 2020, 1–8. https://doi.org/10.1109/IJCNN48605.2020.9207596.

[13] Bruno Dutertre and Leonardo de Moura. 2006. A Fast Linear-Arithmetic Solver for DPLL(T). In Computer Aided Verification. Springer Berlin Heidelberg, 81–94.

[14] Marc Fischer, Milav Lazic, Dasha Drachsler-Cohen, Timon Gehr, Co Zhang, and Martin T. Vechev. 2019. DL2: Training and Querying Neural Networks with Logic. In Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9–15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97), Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 1931–1941. http://proceedings.mlr.press/v97/fischer19a.html.

[15] T. Gehr, M. Mirman, Dasha Drachsler-Cohen, E. Tsvanov, S. Chaudhuri, and M. Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In IJCAI.

[16] Dan R. Ghica and Todd Waugh Ambridge. 2021. Global Optimisation with Constructive Reals. In 36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021, 1–13.

[17] F. W. Grant, J. A. Sharp, M. F. Webster, and X. Zhang. 1996. Sparse matrix representations in a functional language. Journal of Functional Programming 6, 1 (Jan. 1996), 143–170. https://doi.org/10.1017/S095679680000160X Publisher: Cambridge University Press.

[18] Jónathan Heras, Marta Pozza, Maxime Dénes, and Laurence Rideau. 2011. Incidence Simplical Matrices Formalized in Coq/SSReflect. In Intelligent Computer Mathematics (Lecture Notes in Computer Science), James H. Davenport, William M. Farmer, Josef Urban, and Florian Rabe (Eds.). Springer, Berlin, Heidelberg, 30–44. https://doi.org/10.1007/978-3-642-22673-1_3

[19] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10461), Springer, 191–205.

[20] Kai Jia and Martin Rinard. 2021. Exploiting Verified Neural Networks via Floating Point Numerical Error. In Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings (Lecture Notes in Computer Science), Vol. 12913), Springer, 191–205.

[21] Dejan Jovanović and Leonardo de Moura. 2013. Solving non-linear arithmetic. ACM Communications in Computer Algebra 46, 3/4 (Jan. 2013), 104.

[22] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In CAV.

[23] Guy Katz, Clark Barrett, David Dill, Kyle Julian, and Mykel Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. arXiv:1702.01135 [cs] (May 2017). http://arxiv.org/abs/1702.01135 arXiv:1702.01135.

[24] Guy Katz, Derek A. Huang, Duligur Ibring, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Hauze Wu, Aleksandar Željčić, David L. Dill, Mykel J. Kochenderfer, and Clark W. Barrett. 2019. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In CAV 2019, Part I (LNCS, Vol. 11561), Springer, 443–452.

[25] Wen Kokke, Ekaterina Komendantskaya, Daniel Kenitz, Robert Atkey, and David Asfour. 2020. Neural Navigators: Secure by Construction - An Exploration of Refinement Types. In Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12470). Springer, 67–85.

[26] Alexander Kazlov, Ivan Lazarevich, Vasily Shampanskii, Nikolay Lyulyashkin, and Yury Gorbachev. 2021. Neural Network Compression Framework for Fast Model Inference. In Intelligent Computing, Koei Arai (Ed.). Springer International Publishing, Cham, 213–232.
[27] Raghuraman Krishnamoorthi. 2018. Quantizing deep convolutional networks for efficient inference: A whitepaper. CoRR abs/1806.08342 (2018). arXiv:1806.08342
[28] Ori Lahav and Guy Katz. 2021. Pruning and Slicing Neural Networks using Formal Verification. In Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021. 1–10. https://doi.org/10.34727/2021/isbn.978-3-95448-064-4_27
[29] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 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
[30] Elisabetta De Maria, Abdorrahim Bahrami, Thibaud L’Yvonnet, Amy P. Felty, Daniel Gaffé, Annie Ressouche, and Franck Grammont. 2022. On the use of formal methods to model and verify neuronal archetypes. Frontiers Comput. Sci. 16, 3 (2022), 163404.
[31] Grant Olney Passmore. 2021. Some Lessons Learned in the Industrialization of Formal Methods for Financial Algorithms. In Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13047). Springer, 717–721.
[32] Grant O. Passmore, Simon Cruanes, Denys Ignatovich, Dave Atiken, Matt Bray, Elijah Kagan, Kostya Kanishew, Ewen Maclean, and Nicola Mometto. 2020. The Imandra Automated Reasoning System (System Description). In Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II. Vol. 12167. Springer, 464–471.
[33] Connor Shorten and Taghi M. Khoshgoftaar. 2019. A survey on Image Data Augmentation for Deep Learning. Journal of Big Data 6, 1 (July 2019). https://doi.org/10.1186/s40537-019-0197-0 Publisher: Springer Science and Business Media LLC.
[34] Joseph Sill. 1998. Monotonic Networks. California Institute of Technology.
[35] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. 2019. An abstract domain for certifying neural networks. PACMPL 3, POPL (2019), 41:1–41:30. https://doi.org/10.1145/3290354
[36] Natalia Shusarz, Ekaterina Komendantskaya, Matthew L. Daggitt, and Robert Stewart. 2022. Differentiable Logics for Neural Network Training and Verification. https://doi.org/10.48550/arXiv.2207.06741 arXiv:2207.06741 [cs].
[37] Joseph Tassarotti, Koundinya Vajjha, Anindya Banerjee, and Jean-Baptiste Tristan. 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. ACM, 5–17.
[38] Antoine Wehenkel and Gilles Louppe. 2019. Unconstrained Monotonic Neural Networks. In Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada. 1543–1553.
[39] James Wood. 2019. Vectors and Matrices in Agda. https://personal.cis.strath.ac.uk/james.wood.100/blog/html/VecMat.html