Neural Networks in Imandra: Matrix Representation as a Verification Choice*

Remi Desmartin[1], Grant Passmore[2], and Ekaterina Kommendentskaya[1]

[1] Heriot-Watt University, Edinburgh, UK {rhd2000,e.komendantskaya}@hw.ac.uk
[2] Imandra, Austin TX, USA grant@imandra.ai http://www.imandra.ai

Abstract. The demand for formal verification tools for neural networks has increased as neural networks have been deployed in a growing number of safety-critical applications. Matrices are a data structure essential to formalising neural networks. Functional programming languages encourage diverse approaches to matrix definitions. This feature has already been successfully exploited in different applications. The question we ask is whether, and how, these ideas can be applied in neural network verification. A functional programming language Imandra combines the syntax of a functional programming language and the power of an automated theorem prover. Using these two key features of Imandra, we explore how different implementations of matrices can influence automation of neural network verification.

Keywords: Neural networks · Automated reasoning · Formal verification · Functional programming · Imandra.

1 Motivation

Neural network (NN) verification was pioneered by the SMT-solving [9, 10] and an abstract interpretation [1, 6, 17] communities. However, recently claims have been made that functional programming, too, can be valuable in this domain. There is a library [13] formalising small rational-valued neural networks in Coq. 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 [12] uses F^* reals and refinement types for proving robustness of networks trained in Python.

There are several options for defining neural networks in functional programming, ranging from defining neurons as record types [13] to treating them as functions with refinement types [12]. But we claim that two general considerations should be key to any NN formalisation choice of formalisation. Firstly, we must define neural networks as executable functions, because we want to take advantage of executing them in the functional language of choice. Secondly, a generic approach to layer definitions is needed, particularly when we implement complex neural network architectures, such as convolutional layers.

* E.Komendantskaya acknowledges support of EPSRC grant EP/T026952/1.

These two essential requirements dictate that neural networks are represented as matrices, and that a programmer makes choices about matrix formalisation. This extended abstract will explain these choices, and the consequences they imply, from the verification point of view. We use Imandra [14] to make these points, because Imandra is a functional programming language with tight integration of automated proving.

Imandra has been successful as a user-friendly and scalable tool in the FinTech domain [15]. The secret of its success lies in combination of the best features of functional languages and interactive and automated theorem provers. Imandra's logic is based on a pure, higher-order subset of OCaml, and functions written in Imandra are at the same time valid OCaml code that can be executed, or *"simulated"*. Imandra's mode of interactive proof development is based on a typed, higher-order lifting of the *Boyer-Moore waterfall* [3] for automated induction, tightly integrated with novel techniques for SMT modulo recursive functions.

This paper builds upon a recent development of a NN library in Imandra [5], but discusses specifically the matrix representation choices and their consequences.

2 Matrices in Neural Network Formalisation

We will illusrate the functional approach to neural network formalisation and will introduce the syntax of the Imandra programming language [14] by means of an example. When we say we want to formalise neural networks as functions, essentially, we aim to be able to define a NN using just a line of code:

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

where each `layer_i` is defined in a modular fashion.

To see that a functional approach to neural networks does not necessarily imply generic nature of the code, let us consider an example. A *perceptron*, also known as a *linear classifier*, classifies a given input vector $X = (x_1, ..., x_m)$ into one of two classes c_1 or c_2 by computing a linear combination of the input vector with a vector of synaptic weights $(w_0, w_1, ..., w_m)$, in which w_0 is often called an *intercept* or *bias*: $f(X) = \sum_{i=1}^{m} w_i x_i + w_0$. If the result is positive, it classifies the input as c_1 and if negative as c_2. It effectively divides the input space along a hyperplane defined by $\sum_{i=1}^{m} w_i x_i + w_0 = 0$.

In most classification problems, classes are not linearly separated. To handle such problems, we can apply a non-linear function a called an *activation function* to the linear combination of weights and inputs. The resulting definition of a perceptron f is:

$$f(X) = a\left(\sum_{i=1}^{m} w_i x_i + w_0\right) \tag{1}$$

Let us start with a naive prototype of perceptron in Imandra. The Iris data set is a "Hello World" example in data mining; it represents 3 kinds of Iris flowers using 4 selected features. In Imandra, inputs can be represented as a data type:

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

And we define a perceptron as a function:

```
let layer_0 (w0, w1, w2, w3, w4) (x1, x2, x3, x4) =
  relu (w0 +. w1 *. x1 +. w2 *. x2 +. w3 *. x3 +. w4 *. x4)
```

where `*.` and `+.` are *times* and *plus* defined on reals. Note the use of the `relu` activation function, which returns 0 for all negative inputs and acts as the identity function otherwise.

Already in this simple example, one perceptron is not sufficient, as we must map its output to three classes. We use the usual machine learning literature trick and define a further layer of 3 neurons, each representing one class. Each of these neurons is itself a perceptron, with one incoming weight and one bias. This gives us:

```
let layer_1 (w1, b1, w2, b2, w3, b3) f1 =
  let o1 = w1 *. f1 +. b1 in
  let o2 = w2 *. f1 +. b2 in
  let o3 = w3 *. f1 +. b3 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"
```

The second function maps the output of the three neurons to the three specified classes. This post-processing stage often takes a form of an *argmax* or *softmax* function, which we omit.

And thus the resulting function that defines our neural network model is:

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

Although our naive formalisation has some features that we desired from the start, i.e. it defines a neural network as a composition of functions, it is too inflexible to work with arbitrary compositions of layers. In neural networks with hundreds of weights in every layer this manual approach will quickly become infeasible (as well as error prone). So, let us generalise this attempt from the level of individual neurons to the level of matrix operations.

The composition of many perceptrons is often called a *multi-layer perceptron (MLP)*. An MLP consists of an input vector (also called input layer in the literature), multiple hidden layers and an output layer, each layer made of perceptrons with weighted connections to the previous layers' outputs. The weight and biases of all the neurons in a layer can be represented by two matrices denoted by W and B. By adapting equation 1 to this matrix notation, a layer's output L can be defined as:

$$L(X) = a(X \cdot W + B) \tag{2}$$

where the operator $\cdot$ denotes the dot product between X and each row of W, X is the layer's input and a is the activation function shared by all nodes in a layer. As the dot product multiplies pointwise all inputs by all weights, such layers are often called *fully-connected*.

By denoting a_k, W_k, B_k — the activation function, weights and biases of the kth layer respectively, an MLP F with L layers is traditionally defined as:

$$F(X) = a_L[B_L + W_L(a_{L-1}(B_{L-1} + W_{L-1}(...(a_1(B_1 + W_1 \cdot X)))))] \tag{3}$$

At this stage, we are firmly committed to using matrices and matrix operations. And we have two 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.

The first choice was taken in [8] (in the context of dependent types in Coq), in [12] (in the context of refinement types of F*) and in [7] (for sparse matrix encodings in Haskell). The difference between the first and second approaches was discussed in [19] (in Agda, but with no neural network application in mind). The third method was taken in [13] using Coq (though records there were used to encode individual neurons).

In the next three sections, we will systematise these three approaches using the same formalism and the same language, in order to understand the influence they have on neural network verification.

3 Matrices as Lists of Lists

We start with re-using Imandra's `List` library. Lists are defined as inductive data structures. The definition used is identical to OCaml's standard library's:

```
type 'a list =
| []
| :: of 'a * 'a list
```

Imandra holds a comprehensive library of list operations covering a large part of OCaml's standard `List` libary, which we re-use in the definitions below. We start with defining vectors as lists, and matrices as lists of vectors.

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

It is possible to extend this formalisation by using dependent [8] or refinement [12] types to check the matrix size, e.g. when performing matrix multiplication. But in Imandra this facility is not directly available, and we will need to use errors (implemented as a monadic type) to check the matrix sizes.

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 = map2 f x y in
    let tl = map2 f xs ys in
    lift2 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 L_0 distance between two matrices are defined as:

```
let dot_product (a:real matrix) (b:real matrix): ('a, real
    matrix) result =
  Result.map sum (map2 ( *. ) a b)
```

Note that since the output of the function `map2` is wrapped in the monadic `result` type, we must use `Result.map` to apply `sum`. Simlarly, we use standard monadic operations for the `result` monad such as `bind` or `lift`.

A fully connected layer is then defined as a function `fc` that takes as parameters an activation function, a 2-dimensional matrix of layer's weights and an input vector:

```
let activation f w i = (* activation func., weights, input *)
let linear_combination m1 m2 = if (length m1) <> (length m2)
    then Error "invalid dimensions"
```

```
    else map sum (Vec.map2 ( *. ) m1 m2) in
let i' = 1.::i in (* prepend 1. for bias *)
let z = linear_combination w i' in
map f z

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

Listing 1.1: Fully connected layer implementation

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 dot-product of weights and input to account for that.

It is now easy to see that our desired modular approach to composing layers 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 [12]. 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, which clutters the code and adds additional computational checks. Secondly, the matrix definition via the list datatypes makes verification of neural networks very inefficient. This general effect has been already reported in [12], but it may be instructive to look into the problem from the Imandra perspective.

Robustness of neural networks [4] 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 [9], and has resulted in highly efficient SMT solvers specialised on robustness proofs for neural networks [10, 11]. Imandra's waterfall method [14] defines a default flow for the proof search, which starts with unrolling inductive definitions, simplification and rewriting. As a result, proofs of neural network robustness or proofs as in the ACAS Xu challenge, which should not rely on the matrix size induction, stall in Imandra.

There is another mode of proofs available in Imandra: `blast`, a tactic for SAT-based symbolic execution modulo higher-order recursive functions. Blast is an internal custom SAT/SMT solver that can be called explicitely with the appropriate tactic. However, `blast` currently does not support real arithmetic. This requires us to *quantize* the neural networks we use (i.e. convert them to integer weights) and results in a *quantised NN library* [5]. However, even with quantisation and the use of Blast, Imandra fails to scale to the Acas Xu challenge, let alone neural networks used in computer vision.

This also does not come as a surprise: as [10] points out, general-purpose SMT solvers do not scale to NN verification challenges. This is why, the algorithm `reluplex` was introduced in [10] as an additional heuristic to SMT solver algorithms; `reluplex` has since given rise to a domain specific solver Marabou [11]. Connecting Imandra to Marabou may be a promising future direction.

However, there is a silver lining of this method of matrix formalisation. When we formulate verification properties that genuinely require induction, formalisation of matrices as lists does result in more natural, and easily automatable proofs. For example, De Maria et al. [13] formalise in Coq *"neuronal archetypes"* for biological neurons. Each archetype is a specialised kind of perceptron, in which additional functions are added to amplify or inhibit the perceptron's outputs. It is out of scope of this paper to formalise the neuronal archetypes in Imandra, but we take methodological insight from [13]. In particular, [13] shows that there are natural higher-order properties that one may want to verify.

To make a direct comparison, modern neural network verifiers [10, 17] deal with verification tasks of the form "given a trained neural network f, and a property P_1 on its inputs, verify that a property P_2 holds for f's outputs". However, the formalisation in [13] considers properties of the form "any neural network f that satisfies a property Q_1, also satisfies a property Q_2." Unsurprisingly, the former kind of properties can be resolved by simplification and arithmetic, whereas the latter kind requires induction on the structure of f (as well as possibly nested induction on parameters of Q_1).

Another distinguishing consequence of this approach is that it is orthogonal to the community competition for scaling proofs to large networks: usually the property Q_1 does not restrict the size of neural networks, but rather points to their structural properties. Thus, implicitly we quantify over neural networks of any size.

To emulate a property *à la* de Maria et al., in [5] we defined a general network monotonicity property: *any fully connected network with positive weights is* monotone*, in the sense that, given increasing positive inputs, its outputs will also increase*. There has been some interest in monotone networks in the literature [16, 18]. Our experiments show that Imandra can prove such properties by induction on the networks' structure almost automatically (with the help of a handful of auxiliary lemmas). And the proofs easily go through for both quantised and real-valued neural networks. Note that in these experiments, the implementation of weight matrices as lists of lists is implicit – we redefine matrix manipulation functions that are less general but more convenient for proofs by induction.

4 Matrices as Functions

We now return to the verification challenge of ACAS Xu, which we failed to conquer with the inductive matrix representation of the last section. This time we ask whether representing matrices as functions and leveraging Imandra's native proof heuristics can help.

With this in mind, we redefine matrices as functions from indices to values, which gives constant-time (recursion-free) access to matrix elements:

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

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

Listing 1.2: Implementaiton of matrices as functions from indices to values

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 datatypes. We thus recover some functionality given by refinement types in [12].

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).

The resulting function definitions are much more succinct than with lists of lists; take for instance `map2`:

```
let map2 (f: 'a -> 'b -> 'c) (m: 'a t) (m': 'b t) : 'c t =
  function
    | Rows -> rows m
    | Cols -> cols m
    | Value (i,j) -> f (m (Value (i,j))) (m' (Value (i,j)))
```

This allows us to define fully-connected layers:

```
let fc (f: 'a -> 'b) (weights: 'a Matrix.t) (input: 'a Matrix
    .t) =
  let open Matrix in
  function
    | Rows          -> 1
    | Cols          -> rows weights
    | Value (0, j) ->
      let input' = add_weight_coeff input in
      let weights_row = nth_row weights j in
      f (dot_product weights_row input')
    | Value _       -> 0
```

As the biases are included in the `weights` matrix, `add_weight_coeff` prepends a column with coefficients 1 to the input so that they are taken into account.

For full definitions of matrix operations and layers, the reader is referred to [5], but we will give some definitions here, mainly to convey the general style (and simplicity!) of the code. Working with the ACAS Xu networks, a script transforms the original networks into sparse functional matrix representation. For example, layer 5 of one of the networks we used is defined as follows:

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

let layer5_map =
  Map.add (0,0) (1) @@
  Map.add (0,10) (-1) @@
  Map.add (0,29) (-1) @@
  ...
  Map.const 0
```

The sparsity effect is achieved by *pruning* the network, i.e. removing weights that have the smallest impact on the network's performance. The weight's magnitude is used to select those to be pruned. This method, though rather rudimentary, is considered a reasonable pruning technique [**?**]. We do this mainly in order to reduce the amount of computation Imandra needs to perform, and to make the verification problem amenable by Imandra.

With this representation, we are able to verify the properties described in Katz et al. [10] on the pruned networks. This is a considerable improvement compared to the previous section, where the implementation did not allow to verify even pruned networks. It is especially impressive that it comes "for free" with simply changing the underlying matrix representations.

Several factors played a role in automating the proof. Firstly, 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.

is this a good place to say something about connection between maps and the way SMT treats them?

Secondly, 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 memoised. 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.

Although Imandra's simplifier-based automation above could give us results which `blast` could not deliver for the same network, it still did not scale to the original non-quantised (dense) ACAS Xu network. Contrast this with domain-specific verifiers such as Marabou which are able to scale (modulo potential floating point imprecision) to the full ACAS Xu. We are encouraged that the results of this section were achieved without tuning Imandra's generic proof automation strategies, and hopeful that the development of neural-network specific tactics will help Imandra scale to such networks in the future.

5 Real-valued Matrices; Records and Arrays

It is time we turn to the question of encorporating real values into matrices. Section 3 defined matrices as lists of lists; and that definition in principle worked for both interger and real valued matrices. However, we could not use `[@@blast]` to automate proofs when real values were involved; this meant we were restrcted to verifying just integer valued networks. On the other hand, matrices as functions implementation extends to proofs with real valued matrices, however it is not a trivial extension. In the functional implementation given in Listing 1.2 the matrix's value must be of the same type as its dimensions. If defining an integer-valued matrix is easy, this poses a problem for real-valued matrices.

would be nice to show here some code example that gives an idea what the real-valued extension involved

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

- Using an algebraic data type for results of matrix queries: this introduces pattern matching in the implementation of matrix operations. The simplicity and efficiency of the functional implementation is lost.
- 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.
- Use *records* to provide polymorphism and allow matrix to use integer dimensions and real values.

This section focuses on the last two points.

5.1 Proving Termination Over Real Numbers

And for that section, I really can't write much on the implementation detail as I'm still trying to understand parts of it. So if Grant finds the time to write it, good, otherwise can we sweep it under the rug in an honest way? i.e. we give a high-level explaination (we need a floor function to go from reals to integers in order to prove termination) but say we won't go into detail of this proof here?

5.2 Records

Standard OCaml records are available in Imandra, though they do not support functions as fields. They are a specific group of algebraic data types that allow to group together values of different types, offering the possibility of polymorphism for our matrix type.

The approach here is similar to the one in Section 4: matrices are stored as mappings between indices and values, which allows for constant-time access to the elements. However, instead of having the mapping be implemented as a function, here we implement it as a `Map`, i.e. an unordered collection of (key;value) pairs where each key is unique.

```
type 'a t = {
  rows: int;
  cols: int;
  vals: ((int*int), 'a) Map.t;
}
```

Is this a good time to talk about connection to arrays? then what? Remi, see also the slides by Coq people about using arrays for natrices in Coq. May be some comparison will be good here?

We can then use a convenient syntax to create a record of this type:

```
let m = {
  rows = 2;
  cols = 2;
  vals = Map.cons 0.
}
```

Why 0, precisely? to have a matrix of zeros? please explain the motivation for this example. This could be a good moment to emphasise that indices are integers and values are reals, so can we add a comment, and may be have some meaningful real there?

Similarly to the previous implementations, we define a number of useful matrix operations which will be used to define general neural network layer functions. For instance, the definition for the `map2` function is given in Listing 1.3.

```
let rec map2_rec (m: 'a t) (m': 'b t) (f: 'a -> 'b -> 'c) (
    cols: int) (i: int) (j: int) (res: ((int*int), 'c) Map.t)
    : ((int*int), 'c) Map.t =
    let dec i j =
      if j <= 0 then (i-1, cols) else (i,j-1)
    in
    if i <= 0 && j <= 0 then (
      res
    ) else (
      let (i',j') = dec i j in
      let new_value = f (nth m (i',j')) (nth m' (i', j')) in
      let res' = Map.add' res (i',j') new_value in
      map2_rec m m' f cols i' j' res'
    )
```

```
[@@adm i,j]

let map2 (f: 'a -> 'b -> 'c) (m: 'a t) (m': 'b t) : 'c t =
  let rows = max (m.rows) (m'.rows) in
  let cols = max (m.cols) (m'.cols) in
  let vals = map2_rec m m' f cols rows cols (Map.const 0.) in
  {
    rows = rows;
    cols = cols;
    vals = vals;
  }
```

Listing 1.3: Map2 implementation

Compared to the list implementation, this implementation has the benefit of providing constant-time access to matrix elements. However, compared to the implementation of matrices as functions, it uses recursion to iterate over matrix values which results in a high number of case-split. This in turn results in a lower scalability. Moreover we can see in the above function definition that we lose considerable conciseness and readability.

In the end, the main interest of this implementation is its offering polymorphism. In all other regards, the functional implementation is more preferable.

6 Conclusions

Functional programming languages that are tightly coupled with automated reasoning capabilities, like Imandra, offer us the possibility to verify and perform inference with neural networks. In order to do this, implementing matrices and matrices operations is important. We have shown different implementations of matrices and how each implementation influences verification in Imandra; this provides a strong foundation to further develop the Imandra CNN library.

References

1. Ayers, E.W., Eiras, F., Hawasly, M., Whiteside, I.: PaRoT: 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. 12229, pp. 63–84. Springer (2020)
2. Bagnall, A., Stewart, G.: Certifying true error: Machine learning in Coq with verified generalisation guarantees. AAAI (2019)
3. Boyer, R.S., Moore, J.S.: A Computational Logic. ACM Monograph Series. Academic Press, New York (1979)
4. Casadio, M., Komendantskaya, E., Daggitt, M.L., Kokke, W., Katz, G., Amir, G., Refaeli, I.: Neural network robustness as a verification property: A principled case study. In: Computer Aided Verification (CAV 2022). Lecture Notes in Computer Science, Springer (2022)
5. Desmartin, R., Passmore, G., Komendantskaya, E., Daggitt, M.L.: CNN library in Imandra. https://github.com/aisec-private/ImandraNN (2022)

6. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.T.: AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In: S&P (2018)
7. Grant, P.W., Sharp, J.A., Webster, M.F., Zhang, X.: Sparse matrix representations in a functional language. Journal of Functional Programming **6**(1), 143–170 (Jan 1996). https://doi.org/10.1017/S095679680000160X, https://www.cambridge.org/core/journals/journal-of-functional-programming/article/sparse-matrix-representations-in-a-functional-language/669431E9C12EDC16F02603D833FAC31B, publisher: Cambridge University Press
8. Heras, J., Poza, M., Dénès, M., Rideau, L.: Incidence Simplicial Matrices Formalized in Coq/SSReflect. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) Intelligent Computer Mathematics. pp. 30–44. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22673-1_3
9. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: 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. 10426, pp. 3–29 (2017)
10. Katz, G., Barrett, C., Dill, D., Ju-lian, K., Kochenderfer, M.: Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In: CAV (2017)
11. Katz, G., Huang, D.A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljic, A., Dill, D.L., Kochenderfer, M.J., Barrett, C.W.: The Marabou framework for verification and analysis of deep neural networks. In: CAV 2019, Part I. LNCS, vol. 11561, pp. 443–452. Springer (2019)
12. Kokke, W., Komendantskaya, E., Kienitz, D., Atkey, R., Aspinall, D.: Neural networks, 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, pp. 67–85. Springer (2020)
13. Maria, E.D., Bahrami, A., L'Yvonnet, T., Felty, A.P., Gaffé, D., Ressouche, A., Grammont, F.: On the use of formal methods to model and verify neuronal archetypes. Frontiers Comput. Sci. **16**(3), 163404 (2022)
14. Passmore, G.O., Cruanes, S., Ignatovich, D., Aitken, D., Bray, M., Kagan, E., Kanishev, K., Maclean, E., Mometto, N.: 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, pp. 464–471. Springer (2020)
15. Passmore, G.O.: 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, pp. 717–721. Springer (2021)
16. Sill, J.: Monotonic Networks. California Institute of Technology (1998)
17. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: An abstract domain for certifying neural networks. PACMPL **3**(POPL), 41:1–41:30 (2019). https://doi.org/10.1145/3290354
18. Wehenkel, A., Louppe, G.: 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. pp. 1543–1553 (2019)

19. Wood, J.: Vectors and Matrices in Agda (Aug 2019), https://personal.cis.strath.ac.uk/james.wood.100/blog/html/VecMat.html