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
[2] Imandra Inc. Austin TX, USA `lncs@springer.com`
http://www.springer.com/gp/computer-science/lncs
[3] ABC Institute, Rupert-Karls-University Heidelberg, Heidelberg, Germany
`{abc,lncs}@uni-heidelberg.de`

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: Matrices in Neural Network Formalisation

Neural network (NN) verification was pioneered by the SMT-solving [?,?] and an abstract interpretation [?,?,?] communities. However, recently claims have been made that functional programming, too, can be valuable in this domain. There is a library [?] formalising small rational-valued neural networks in Coq. A more sizeable formalisation called MLCert [?] 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 [?] 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 [?] to treating them as functions with refinement types [?]. 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

[⋆] E.Komendantskaya acknowledges support of EPSRC grant EP/T026952/1.

advantage of simulating/running/executing them in the functional language of choice. Secondly, a generic approach to layer definitions is needed, particularly when we implement convolutional layers. We will illusrate these two points and will introduce the syntax of the Imandra programming language [?] 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:

Listing 1.1: Desirable syntax for CNN definition

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

Perceptrons. 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.

Neural Network Layers. 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:

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

2 Matrix as Lists of Lists

2.1 using Sized Lists or Vectors

Grant et al. ([1]) proposes 4 sparse list-based matrix implementations. They use an array-as-trees representation which allows to optimise for sparse arrays (subtrees where all the leafs are 0 are replaced by a 0-leaf).

Binary trees and lists of row-fragments: binary tree array of sparse Vectors defined as $[(\mathrm{Int},\ [\mathrm{Double}])]$

A generalised envelope scheme: matrix is cut up in sections A quadtree scheme: Triangular matrix is split up in 2 triangular and a rectangular one. A standard quadtree structure is used for the rectangular matrix.

A Two-copy list of row-segments scheme: list of row-segments and list of column-segments in order to iterate over columns easily. Con: 2x more space is used; can be used to improve the 2 first previous methods (quadtree already bidimensional)

Pros of sparse list-based matrix representation: optimised for sparse matrices. Optimised for the specific operation considered in the paper (solving of linear systems of equations using a Cholesky scheme)

2.2 Refined Types

Coq/Mathcomp/SSReflex: the size of the matrix is defined as a refinement type and used to check that matrix have the appropriate size in multiplications Heras et al [2] discuss implementation; correctness is checked at compile-time.

This is also used in Starchild/Lazuli [3].

2.3 Arrays

Grant et al. [1] mentions using Haskell mutable arrays which are implemented using monadic operations. They stress that using a mutable array allows for modifying the array in place (thus saving memory), but it introduces "extra programming difficulties"; i.e. the use of monads makes the code less clear (as is the case in 2.4).

In our case, since IML is pure, there is no access to mutable arrays.

2.4 Monadic Operations to Check Size

Allows to check for valid matrix sizes when no refinement types are available. Intuitive first representation.

Drawback: introduces a lot of pattern matching, so a lot of split cases which increases the size of the program to check exponentially

Drawback: no optimisation for sparse matrices

3 Matrix as Functions

Imandra implementation; matrices are defined as total functions mapping indices to values. Theory of uninterpreted functions.

Woods [4] discusses the benefits of implementing matrices as functions in Agda.

4 Matrix as Maps

References

1. 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
2. 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
3. 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, vol. 12470, pp. 67–85. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-64437-6_4, series Title: Lecture Notes in Computer Science
4. Wood, J.: Vectors and Matrices in Agda (Aug 2019), https://personal.cis.strath.ac.uk/james.wood.100/blog/html/VecMat.html