Neural Networks, Secure by Construction
An Exploration of Refinement Types[⋆]

Wen Kokke[1,2], Ekaterina Komendantskaya[1], Daniel Kienitz[1], Robert Atkey[3], and David Aspinall[2]

[1] Heriot-Watt University, Edinburgh, UK {ek19,dk50}@hw.ac.uk
[2] University of Edinburgh, Edinburgh, UK {wen.kokke,david.aspinall}@ed.ac.uk
[3] Strathclyde University, Glasgow, UK robert.atkey@strath.ac.uk

Abstract. We present StarChild and Lazuli, two libraries which leverage refinement types to verify neural networks, implemented in F* and Liquid Haskell. Refinement types are types augmented, or *refined*, with assertions about values of that type, *e.g.*, "integers greater than five", which are checked by an SMT solver. Crucially, these assertions are written in the language itself. A user of our library can refine the type of neural networks, *e.g.*, "neural networks which are robust against adversarial attacks", and expect F* to handle the verification of this claim for any specific network, without having to change the representation of the network, or even having to learn about SMT solvers.
Our initial experiments indicate that our approach could greatly reduce the burden of verifying neural networks. Unfortunately, they also show that SMT solvers do not scale to the sizes required for neural network verification.

Keywords: Neural Networks · Verification · Refinement Types

1 Introduction

Deep neural networks—or simply *neural networks*—is an umbrella term for a range of machine learning algorithms that, given numeric data instances as an input, construct a *non-linear function* or *classifier* that separates these data instances into classes. When a suitable classifier is found, it can be used to classify new, unseen data—or at least, that's the hope. Data instances can be pixel data for images, numeric encodings of the words from a lexicon for text analysis, or generally any n features of interest, viewed as a point in an n-dimensional real space. There are numerous applications of neural networks: in computer vision, natural language processing, data mining, to name but a few. As neural networks move into domains where safety and security are important—*e.g.*, autonomous cars, conversational agents, governance—the problem of their verification comes to the forefront.

[⋆] The work was funded by the National Cyber Security Center, UK. Grant *SecConn-NN: Neural Networks with Security Contracts—towards lightweight, modular security for neural networks.*

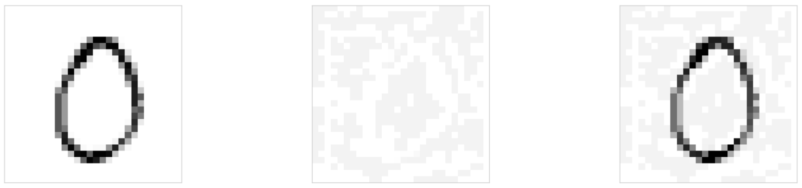


Fig. 1: (Left) Image from MNIST [16] dataset, which is correctly classified as 0 *by a given neural network. (Center) A small perturbation applied to the image. (Right) Resulting noisy image classified by the same neural network as a* 3 *with* 92% *confidence.*

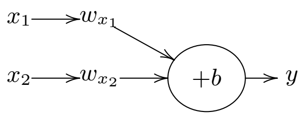
Neural network verification is a notoriously difficult problem. Firstly, neural networks rely on data for training, testing, and often for verification. This data may be incomplete, noisy, or deliberately *poisoned*. Secondly, finding a suitable classifier is a mathematically complex task. There is a continuum of suitable classifiers in a continuous real space, and the search space may be prohibitively large, and an optimal classifier may not even exist. Finally, neural networks are difficult to interpret. Even if a reasonably accurate classifier is found, we do not understand all its latent properties. This is particularly true for classifiers that work with data of high dimensionality.

The very features that we value in neural networks (adaptivity and the ability to generalise from noisy data) becomes a source of safety and security threats. Neural networks are known to be vulnerable to *adversarial attacks* [25, 10, 19, 21, 22] (specially crafted inputs that can create an unexpected and possibly dangerous output) and suffer from *catastrophic forgetting* [20].

One approach to the verification of complex problems is *lightweight verification*, which means to:

1. verify only the properties that *matter* [9],
2. embed verification in the implementation, and
3. employ proof automation where possible.

In neural network verification, one property that matters is *adversarial robustness*, commonly characterised as the deviation in the neural network's outputs given perturbations of its inputs, checked for some set of inputs [23, 11, 13]. For datasets with relatively low inner-class variation, like MNIST [16], we can pick our sample images either randomly or by hand, and define perturbations using some valid transformations like rotation, scaling, and translation. For example, we could pick the image on the left of Figure 1 as a sample image for the class zero, and verify whether, given a certain range of perturbations defined by a suitable distance function, we can guarantee that the perturbed image is still classified as 0. Such method would not cover unanticipated perturbations, *e.g.*, since we did not think of noise, the image on the right of Figure 1 is not covered by our safety guarantees. This is not the only possible interpretation of the



x_1	x_2	$x_1 \wedge x_2$
1	1	1
1	0	0
0	1	0
0	0	0

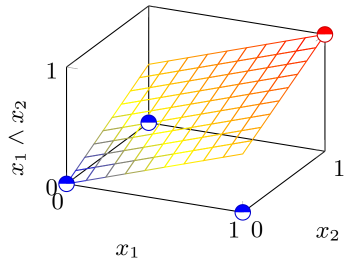


Fig. 2: (Left) Perceptron shown graphically as a neural network. (Center) Dataset for perceptron. (Right) Dataset as points in the three-dimensional space, with a linear classifier for the data.

"neural network verification problem", but it is by far the most common. We will therefore use it throughout the paper.

We are primarily interested in exploring the space of solutions for (2) and (3). Since neural networks are "just" functions, we seek to embed verification constraints on inputs and outputs in the types of these functions, and then use the facilities of refinement type checking—with SMT solver integration—to automate all tedious proofs. In this paper, we explore this space using F* [24] and Liquid Haskell [27] and test whether contemporary, off-the-shelf programming language technologies are suitable for neural network verification, and to analyse the benefits and limitations of using refinement types. We hope the reader will find this study useful, by employing our ideas, avoiding the pitfalls we encountered, and perhaps filling the gaps in contemporary programming language technologies.

1.1 Example: Verifying the AND-gate

Let's use a simple example to illustrate the use of our library: a perceptron for the logical AND-gate [17]. It has two inputs, a single, fully-connected layer, and one output, and its training set is the truth table for Boolean conjunction (see Figure 2).

The *perceptron* is a gradient descent algorithm that approximates the linear function:

$$\begin{aligned}&\text{neuron} : (x_1 : \mathbb{R}) \to (x_2 : \mathbb{R}) \to (y : \mathbb{R})\\ &\text{neuron}\ \ x_1\ x_2 = b + w_{x_1} \times x_1 + w_{x_2} \times x_2\end{aligned}$$

that separates the data points into two classes, as shown in Figure 2. The constants w_{x_1} and w_{x_2} are called the *weights* of the neuron, and b its bias. The gradient descent algorithm searches for suitable values for these constants, e.g.:

$$\text{neuron}\ \ x_1\ x_2 = -0.9 + 0.5x_1 + 0.5x_2$$

Often, perceptrons involve an activation function, which is applied to the result of the linear function. Here, we use the *threshold function* S. We discuss other activation functions in Section 4.

$$S\ x = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

We can refine the output type of our new neuron function, as S only ever returns 0 or 1:

$$\begin{aligned} &\text{neuron} : (x_1 : \mathbb{R}) \to (x_2 : \mathbb{R}) \to (y : \mathbb{R}\ \{y = 0 \lor y = 1\}) \\ &\text{neuron}\ x_1\ x_2 = S\ (-0.9 + 0.5x_1 + 0.5x_2) \end{aligned}$$

Let's verify that the neural network returns the "correct" values for inputs which lie within some distance ϵ of 1 and 0. Let's call these values *truthy* and *falsey*:

$$\begin{aligned} \text{truthy}\ x &= |1 - x| \leq \epsilon \\ \text{falsey}\ x &= |0 - x| \leq \epsilon \end{aligned}$$

We can request that F* checks whether our neural network is correct by refining the type of neuron, *e.g.*, by requiring that the output be 1 if both inputs are truthy. If neuron does not satisfy this property, test will not type check:

$$\begin{aligned} &\text{test}\ :\ (x_1 : \mathbb{R}\ \{\text{truthy}\ \ x_1\}) \to (x_2 : \mathbb{R}\ \{\text{truthy}\ \ x_2\}) \to (y : \mathbb{R}\ \{y = 1\}) \\ &\text{test}\ =\ \text{neuron} \end{aligned}$$

The user can implement the network *directly* in F*. Alternatively, if they have a pre-existing neural network in, *e.g.*, Python, they can *export* the network to F*, as a Python library to export networks is included in both StarChild and Lazuli. For instance, we can use a Python library to find a suitable function for the data in Figure 2, and export our model to F* to obtain the following code:

```
val model : network (*with*) 2 (*inputs*) 1 (*output*) 1 (*layer*)
let model = NLast // ← makes single-layer network
  { weights    = [[0.5R]; [0.5R]]
  ; biases     = [−0.9R]
  ; activation = Threshold }
```

Let's verify that it is correct for, *e.g.*, $\epsilon = 0.1$, in F*:

```
let eps      = 0.1R
let truthy x = 1.0R - eps ≤ x && x ≤ 1.0R + eps
let falsey x = 0.0R - eps ≤ x && x ≤ 0.0R + eps

val test : (x₁ : ℝ{truthy x₁}) → (x₂ : ℝ{truthy x₂})
        → (y  : vector ℝ 1 {y ≡ [1.0R]})
let test x₁ x₂ = run model [x₁; x₂]
```

Refinement types, used in this manner, seem to be a natural fit. The "burden" of verifying the AND-gate in our approach is minuscule. Once written, the user can reuse the code for `test` to verify different neural networks that use similar verification conditions, and develop a codebase of reusable verification conditions.

As a benefit of using F*, any model specified using StarChild, and any other F* program, is usable in refinements, and F* takes care of the translation to the SMT logic for us! For instance, when F* checks the function `test`, it passes the definition and the refinements on the inputs and output to the SMT solver, and only accepts the function if the SMT solver does. It *does not* check the networks output for all inputs within distance ϵ—this wouldn't be feasible, as there are uncountably many, and even accounting for the maximum precision of floating-point numbers, the search space is vast.

F* translates programs to the SMT logic by normalising them, translating constructs to their SMT equivalents where possible, and keeping the rest as uninterpreted functions. For instance, `test` normalises to:

```
let test x₁ x₂ = if x₁×0.5R + x₂×0.5R − 0.9R ≥ 0.0R then 1.0R else 0.0R
```

The normalised version can be translated directly to the SMT logic, together with the type refinements for `test`. This generates the following SMT query—simplified for readability—in SMTLIB2 Lisp [5]:

```
(define-fun neuron ((x₁ Real) (x₂ Real)) Real
  (ite (>= (- (+ (* x₁ 0.5) (* x₂ 0.5)) 0.9) 0.0) 1.0 0.0))
(define-fun truthy ((x Real)) Bool (and (<= 0.9 x) (<= x 1.1)))
(assert (∀ ((x₁ Real) (x₂ Real))
  (=> (and (truthy x₁) (truthy x₂)) (= (neuron x₁ x₂) 1.0))))
(check-sat)
```

As it turns out, this particular query is satisfiable, which you can verify with your favourite SMTLIB2-compatible solver. Therefore, our neural network is robust around truthy inputs!

1.2 Contributions

We make several contributions:

- We introduce two libraries, StarChild [4] for F*, and Lazuli[5] for Liquid Haskell. These libraries allow users to conveniently and modularly define and verify neural networks (Section 2).
- We illustrate that both F* and Liquid Haskell are suitable for the lightweight verification of neural networks (Section 2).
- We describe an approach for translating Keras [6] models, *e.g.*, generated in Python, to StarChild and Lazuli (Section 2.2).
- We describe an approach for automating proofs involving non-linear activation functions, by piecewise-linearisation. SMT queries using non-linear functions such as the exponential function are not generally supported, and problems involving such functions are generally undecidable. However, all deep neural networks use non-linear activation functions, such as Sigmoid or Softmax (Section 4)
- We show that both training and testing using piecewise-linear approximations of non-linear activation functions is possible, and results in only a negligible decrease in performance (Section 4).

[4] https://github.com/wenkokke/starchild
[5] https://github.com/wenkokke/lazuli

– Finally, we describe several problems that we believe cannot be overcome without substantial improvements in both the programming languages, *e.g.*, F* and Liquid Haskell, and in SMT solvers. These are problems of scale, and limitations that arise from the incomplete implementation of real-valued expressions in F*, and the lack of normalisation of refinements in Liquid Haskell. We suggest possible solutions for the future (Section 5).

Neural network verification is a growing area of research, with several tools on the market, e.g. Marabou [13], ERAN [23], DLV [11], PAROT [3], to name a few. It is not our goal to produce another competing tool, hence the missing benchmarking against these. Instead, our goal is to establish programming language principles for incorporating these tools into a more abstract framework, which may open ways of embedding neural net verification into future multi-component projects.

2 An Overview of StarChild

Neural networks are functions on vectors of real numbers. Hence, the StarChild library consists mostly of an F* implementation of basic linear algebra (implemented in `StarChild.LinearAlgebra`). A second module contains an implementation of dimension-safe neural networks, following Grenade[6] and the "dependently-typed" Haskell bindings for TensorFlow[7][8] (implemented in `StarChild.Network`).

The linear algebra module defines the types of length-indexed real vectors and matrices, using F*'s implementation of real numbers (implemented in `FStar.Real`), and using refinements of F*'s implementation of lists for both vectors and matrices, where the refinement adds a length-index.

The module further defines standard operations on vectors and matrices: maps and folds, the dot product, and matrix multiplication (see Figure 3). We reuse the list implementations of these functions when possible, but often F* needs us to redefine functions, *e.g.*, `map1`, to verify the length-index.

Finally, the module defines common distance metrics on vectors, which can be used in verification constraints. However, not all distance metrics can be represented in the SMT logic. For instance, Euclidean distance uses the square root function, which is non-linear. Instead, we opt to use the squared Euclidean distance (see Figure 3).

The neural network module defines the structure of neural networks. A neural network is a non-empty list of layers, where the number of outputs of each layer matches the number of inputs of the next layer. The network type has three parameters—the number of inputs, outputs, and layers. Just like with lists, there are two ways to construct a network. `NLast` defines a single-layer network, whose number of inputs and outputs correspond to those of the layer. `NStep` adds a layer to the front of a network, where the number of inputs of new layer becomes the

[6] https://github.com/HuwCampbell/grenade
[7] https://hackage.haskell.org/package/tensor-safe
[8] https://github.com/helq/tensorflow-haskell-deptyped

```
type vector 'a n   = v:list 'a {length v ≡ n}
type matrix 'a r c = vector (vector 'a c) r

val dot : #n:ℕ // Dot product
       → xs:vector ℝ n → ys:vector ℝ n → ℝ
let dot #n xs ys = sum (map2 (fun x y → x × y) xs ys)

val vAv : #n:ℕ // Vector addition
       → xs:vector ℝ n → ys:vector ℝ n → vector ℝ n
let vAv #n xs ys = map2 (fun x y → x + y) xs ys

val vXm : #r:ℕ → #c:ℕ // Vector-matrix multiplication
       → xs:vector ℝ r → yss:matrix ℝ r c → vector ℝ c
let rec vXm #r #c xs yss = match xs, yss with
  | [], [] → replicate 0.0R
  | (x :: xs), (ys :: yss) →
    vAv #c (scale #c x ys) (vXm #(r - 1) #c xs yss)

val mXm : #i:ℕ → #j:ℕ → #k:ℕ // Matrix-matrix multiplication
       → matrix ℝ i j → matrix ℝ j k → matrix ℝ i k
let mXm #i #j #k xss yss = map (fun xs → vXm #j #k xs yss) xss

val sed : #n:ℕ // Squared Euclidean distance
       → xs:vector ℝ n → ys:vector ℝ n → ℝ≥0
let sed #n xs ys =
  sum≥0 #n (map2 #ℝ #ℝ #ℝ≥0 #n (fun x y → (x − y) × (x − y)) xs ys)
```

Fig. 3: Linear algebra functions in StarChild.

number of inputs of the network, and the number of outputs of the new layer has to correspond to the old number of inputs of the network:

```
type network (i:ℕ>0) (o:ℕ>0) : n:ℕ → Type =
  | NLast : l:layer i o → network i o 1
  | NStep : #n:ℕ>0 → #h:ℕ>0
         → l:layer i h → ls:network h o n → network i o (n + 1)
```

We use $\mathbb{N}_{>0}$ to denote the refined type of positive natural numbers, and similarly, $\mathbb{R}_{>0}$ and $\mathbb{R}_{\geq 0}$ to denote the positive and non-negative real numbers.

Each fully-connected layer consists of a matrix of weights, whose dimensions correspond to the number of inputs and outputs of the layer, a vector of biases, whose length corresponds to the number of outputs of the layer, and the name of an activation function:

```
type layer (i:ℕ>0) (o:ℕ>0) =
  { weights    : matrix ℝ i o
  ; biases     : vector ℝ o
  ; activation : activation }
```

```
val lsigmoid : ℝ → ℝ
let lsigmoid x = let v = 0.25R × x + 0.5R in
                 if v < 0.0R then 0.0R else
                 if 1.0R < v then 1.0R else v

val lexp : ℝ → ℝ_{>0}
let lexp x = if x ≤ − 1.0R then 0.00001R else
             if x ≥ 1.0R then 5.898R × x − 3.898R else x + 1.0R

val norm : #n:ℕ_{>0} → vector ℝ_{>0} n → vector ℝ n
let norm #n xs = map1 #ℝ_{>0} #ℝ #n (fun x → x / sum_{≥0} #n xs) xs

val lsoftmax : #n:ℝ_{>0} → vector ℝ n → vector ℝ n
let lsoftmax #n xs = norm (map1 #ℝ #ℝ_{>0} #n lexp xs)
```

Fig. 4: Naive piecewise-linear approximations of the Sigmoid and Softmax functions in the StarChild library.

Our current implementation supports four common activation functions:

```
type activation : Type =
  | Linear  // $\text{linear}(x) = x$
  | ReLU    // $\text{relu}(x) = \max(0, x)$
  | Sigmoid // $\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$
  | Softmax // $\text{softmax}(\bar{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}}$
```

The linear and ReLU functions are straightforward to define, although the `FStar.Real` module is rather sparse, and lacks functions for, *e.g.*, minimum, maximum, negation, *etc.*:

```
val linear : ℝ → ℝ
let linear x = x // i.e. identity function
val relu : ℝ → ℝ
let relu x = if x ≤ 0.0R then x else 0.0R
```

However, the Sigmoid and Softmax functions are non-linear functions, and cannot be translated directly to the SMT logic. Our solution is to use piecewise-linear approximations of these functions. Since F* does not allow us to fine-tune the translation to the SMT logic, we implement these directly in F*. In Figure 4, we present two naive implementations of piecewise-linear approximations for the Sigmoid and Softmax functions. We discuss a more principled approach to generating linear approximations in Section 4.

To run a StarChild network, we simply run each layer successively, feeding the output of one layer into the next:

```
val run : #i:ℕ_{>0} → #o:ℕ_{>0} → #n:ℕ_{>0}
       → ls:network i o n → xs:vector ℝ i
       → Tot (vector ℝ o) (decreases n)
let rec run #i #o #n ls xs = match ls with
```

```
  | NLast l    → run_layer l xs
  | NStep l ls → run ls (run_layer l xs)
```

We annotate the function with a totality annotation, which lets F* verify the recursion terminates by checking that the number of layers decreases.

We run a layer by performing the computations described in Section 1.1: we multiply the inputs by the weights, add the bias, and run the activation function:

```
val run_layer : #i:ℕ>0 → #o:ℕ>0
            → l:layer i o → xs:vector ℝ i → vector ℝ o
let run_layer #i #o l xs =
  run_activation #o l.activation (vAv #o l.biases (vXm #i #o xs l.weights))
```

Finally, we run an activation function by matching the name, *e.g.*, `Sigmoid`, up with the appropriate definition, *e.g.*, `lsigmoid`:

```
val run_activation : #n:pos → a:activation → xs:vector ℝ n → vector ℝ n
let run_activation #n a xs =
  match a with
  | Linear  → xs
  | ReLU    → map1 relu xs
  | Sigmoid → map1 lsigmoid xs
  | Softmax → lsoftmax xs
```

2.1 A Note on Lazuli

The Liquid Haskell counterpart to StarChild, Lazuli, follows a similar architecture, and shares the module and function names whenever possible. Any differences are due to quirks of F* or Liquid Haskell.

When implementing dimension-safe vector arithmetic in Liquid Haskell, it is convenient to store the dimensions of a vector or matrix in the structure itself, hence, in Lazuli, vectors and matrices are refined record types. For instance, a vector is a record which stores a list and an integer, with a type refinement requiring that that integer is exactly equal to the length of the list.

Liquid Haskell allows us to fine-tune the translation of functions to the SMT language, hence, if the user wants to, they could translate the standard Softmax function to the linearised Softmax *only* during verification. This has consequences for the safety guarantees, however, as the verified network no longer corresponds *exactly* to the executed network.

Finally, Liquid Haskell does not support normalisation prior to the translation to the SMT logic. Instead, Liquid Haskell supports *refinement reflection* [28], in which Haskell functions are translated to SMT equalities which encode their reduction behaviour. This offloads the burden of normalisation to the SMT solver. Unfortunately, SMT solvers perform exploratory search, in which they use these equations in *both* directions, *i.e.*, they expand as well as reduce. Hence, they are much less efficient at reduction, and consequently, at the time of writing Lazuli is significantly slower than StarChild.

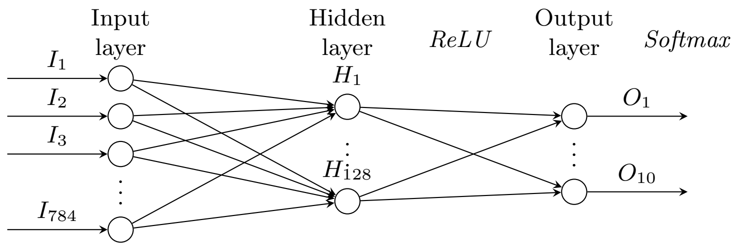
2.2 The Convenience of Keras Models

We don't have any illusions that training networks in F* or Liquid Haskell will be the preferred method, or even feasible, in the near future. Therefore, it is important to integrate our libraries with existing methods. For this reason, we implemented a Python library for converting Keras [6] models to StarChild and Lazuli, which we bundle with StarChild and Lazuli as `convert.py`.

3 Verifying a "Real" Example: MNIST

In this section, we describe our experiences using StarChild to verify a neural network trained on MNIST.

The MNIST dataset contains 28×28 images of the handwritten digits "0" to "9". Hence, an input consists of 748 pixels, and an output is—conventionally—a probability distribution over the 10 classes, created by the Softmax function. This leaves us to determine the number of hidden layers, their sizes, and their activation functions. For instance, we could opt for a 128-node hidden layer using the ReLU activation function:



Unfortunately, this model has $784 \times 128 + 128 + 128 \times 10 + 10 = 101770$ constant parameters and 784 input parameters. Worse, it has 3 fully-connected layers, meaning that each input parameter occurs at least $128 \times 10 = 1280$ times in the SMT query, and constant parameters occur several times in accordance to the layer they are in. This is a *huge* query from an SMT solving perspective, and it would overwhelm any SMT solver. However, this is not a large network from a machine learning perspective. We discuss this matter further in Section 5.

For now, we seek to make verification with an SMT solver tractable. One option is to reduce the dimensionality of the input, and reduce the size of the network. If used with care, this usually only leads to modest decreases in model accuracy. We use *principal component analysis* (PCA) to reduce the size of the input vectors to 25, and reduce the size of the hidden layer to 10. This model has *far fewer* parameters, $25 \times 10 + 10 + 10 \times 10 + 10 = 370$, yet it only suffers a loss of 2 percentage points in test accuracy (see Table 9). Note that verifying the correctness of the smaller model gives us no formal guarantees about the correctness of the larger model. Hence, using this approach, we are forced to deploy the smaller, less accurate model. Figure 5 shows the F* code for the smaller MNIST network, imported from Keras using the library described in Section 2.2.

Unlike in Section 1.1, vectors in Figure 5 are wrapped in an assertion (**let** v = … **in** **assert_norm** (length v = n); v). There are two assertion keywords, **assert** and **assert_norm**. These assertions have no runtime significance. Instead, one can think of them as functions with the refined type (b:bool {b ≡ true}) → (). That is to say, assertions take an argument of type bool and verify, using an SMT solver, that it is true.

By default, terms are translated to the SMT logic *unnormalised*, similar to Liquid Haskell (see Section 2.1). After all, terms may grow enormously through normalisation. Using **assert_norm** forces F* to normalise terms before querying the SMT solver. Without it, F* offloads the burden of term reduction to the SMT solver. Unfortunately, SMT solvers do exploratory search, and are much less efficient at reduction. Worse, F* encodes a notion of fuel into translated terms, meaning function definitions can only be unfolded a set number of times, determined by the command-line argument --max-fuel (default 8). Beyond that, programs fail to type check.

Let's verify the model is robust for class "0" in an ϵ-ball $\mathbb{B}(\hat{x})$ around a sample input $\hat{x}$, $\mathbb{B}(\hat{x}, r) = \{x \in \mathbb{R}^n : ||\hat{x} - x||_2 \leq r\}$. First, we pick an input vector representing the digit "0", and convert it to F*:

```
val sample_in : vector ℝ 25
let sample_in = let v = [7.394R; −0.451R; …; 0.199R]
                in assert_norm (length v = 25); v
```

Then, we run the Keras model on the input, and convert the output to F*:

```
val sample_out: vector ℝ 10
let sample_out = let v = [0.998R; 0.000R; …; 0.000R]
                 in assert_norm (length v = 10); v
```

For readability, we elide several elements from each vector, and limit the precision of the floating-point numbers.

With these two definitions in hand, we can define our verification condition. The idea is that, for all inputs within a certain distance ϵ_1 from our sample input, the neural network output should be within a certain ϵ_2 from the sample output. Let $\epsilon_1 = 0.01$ and $\epsilon_2 = 1$:

```
let _ = assert_norm (∀ (x:vector ℝ 25). (sed #25 sample_in x < 0.01R)
   ⟹ (sed #10 sample_out (run m x) < 1.0R))
```

Note that the function sed (squared Euclidean distance) is defined in Figure 3. While type checking, F* verifies that our verification condition holds. Crucially, it wouldn't be possible to verify this by testing.

Once again, the "burden" of verification in our approach is rather small, as it takes only a handful of lines of code to formulate the verification conditions, and the code which checks them. Unfortunately, even for this modest model, verification of complex conditions takes an *infeasibly long* amount of time. We address this problem in Section 5.

```
val layer_0 : layer 25 10
let layer_0 =
  { weights = (let v = [ (let v = [−0.64R; 0.19R; …; 0.54R; 0.78R]
                            in assert_norm (length v = 10); v)
                         ; (let v = [0.79R; 0.53R; …; −1.00R; 0.82R]
                            in assert_norm (length v = 10); v)
                         …
                         ; (let v = [−0.33R; −0.44R; …; −0.20R; −0.04R]
                            in assert_norm (length v = 10); v) ]
               in assert_norm (length v = 25); v)
  ; biases = (let v = [0.15R; 1.28R; …; 1.03R; 0.32R]
              in assert_norm (length v = 10); v)
  ; activation = ReLU }

val layer_1 : layer 10 10
let layer_1 =
  { weights = (let v = [ (let v = [0.00R; −0.87R; …; −0.99; 0.26R]
                            in assert_norm (length v = 10); v)
                         ; (let v = [−0.50R; −1.28R; …; 0.65R; 0.62R]
                            in assert_norm (length v = 10); v)
                         …
                         ; (let v = [0.50R; −0.49R; …; −0.73R; −0.34R]
                            in assert_norm (length v = 10); v) ]
               in assert_norm (length v = 10); v)
  ; biases = (let v = [−0.41R; −0.41R; …; 1.08R; 0.15R]
              in assert_norm (length v = 10); v)
  ; activation = Softmax }

val model : network 25 10 2
let model = NStep layer_0 (NLast layer_1)
```

Fig. 5: StarChild model generated from Keras.

4 Piecewise-Linear Approximations Made Easy

In this section, we discuss non-linear activation functions, and automatic linearisation. Deep neural networks require the use of non-linear functions between each layer—the composition of two linear functions is itself a linear function, and hence any deep neural network which uses only linear activation functions is equivalent to a shallow neural network.

Unfortunately, SMT solvers do not generally support non-linear arithmetic, and where they do, the solvers are slower and less reliable. At the time of writing, F* uses the Z3 solver [18]. Z3 uses Dual Simplex [7] to solve linear real arithmetic. It also supports a fragment of non-linear real arithmetic—specifically, multiplications—and solves this using a conflict resolution procedure based on cylindrical algebraic decomposition [12]. However, the addition of multiplication

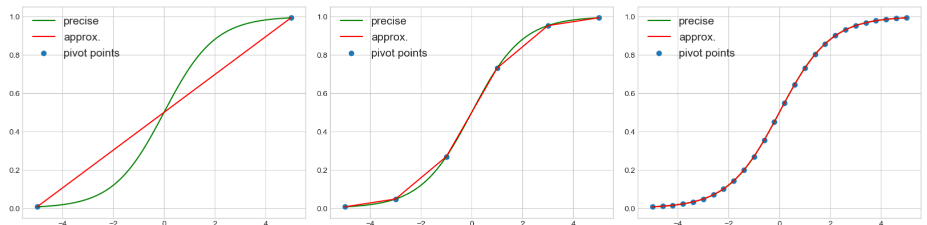


Fig. 6: Linearisation of the Sigmoid function over the interval $I = [-5, 5]$ with $n = 1$, $n = 5$, and $n = 25$ line segments.

is not enough to cover the non-linear activation functions used in deep learning, which often use exponents, logarithms, and trigonometric functions. The only solver we are aware of that supports these functions out of the box is MetiTarski [2]. However, the MetiTarski documentation reads “Beyond 4 or 5 continuous variables, there is very little hope for MetiTarski in finding a proof.” Since our smallest possible “real” problem involves 25 continuous variables, we have very little hope that MetiTarski will prove useful to us.

We approximate non-linear activation functions using piecewise-linear approximations, *i.e.*, several connected line segments. We refer to this as “linearisation”. For instance, in Figure 4 we used two handwritten piecewise-linear approximations for the Sigmoid and the exponential functions. This approach is a little crude, and manual linearisation is time consuming. Instead, we have developed an algorithm for automatic linearisation of a function $\sigma : \mathbb{R} \to \mathbb{R}$ over an interval I using n line segments:

1. We split the interval I into n equal-sized sub-intervals $I_1, \ldots, I_n$.
2. For each sub-interval I_i:
   (a) Let $l_i = \min I_i$ and $u_i = \max I_i$.
   (b) We draw a line segment of the form $f_i(x) = m_i x + b_i$, with slope m_i and y-intercept b_i, from the minimum $(l_i, \sigma(l_i))$ to the maximum $(u_i, \sigma(u_i))$.
3. Finally, we connect all line segments f_i. The result is a piecewise-linear approximation for σ over the interval I.

The parameter n determines the granularity. In Figure 6, we show the linear approximation of the Sigmoid function for different values of n.

How should a piecewise-linear approximation behave outside of the interval I? We have three simple options:

1. We can extrapolate the first and last line segments beyond the interval boundaries.
2. We can return the minimum point, $\sigma(\min I)$, for inputs below the interval, and return the maximum point, $\sigma(\max I)$, for inputs above the interval.
3. We can combine (1) and (2). We start by extrapolating, following (1), and allow the user to specify lower and upper bounds, where we switch to returning the constant minimum and maximum, following (2).

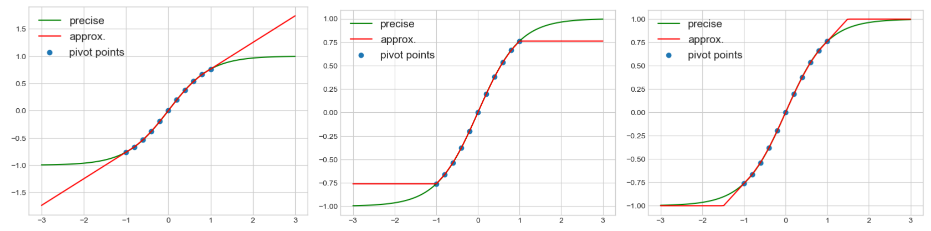


Fig. 7: Linearisation of the tanh*-function over the interval* $[-1, 1]$ *with* $n = 10$ *with three different bounding methods: extrapolation, constant values, and the user-defined combination.*

The first option is unsound, as it may result in cases where the codomain of the piecewise-linear approximation is not a subset of the codomain of the approximated function. For instance, the piecewise-linear approximation of the exp-function may return values < 0 for a sufficiently small input. The second option is sound, albeit a bit crude. The third option combines the best of (1) and (2), but requires manual tweaking. In Figure 7, we show examples of these methods for the tanh-function.

Piecewise-linear functions are not continuously differentiable, as they are non-differentiable at each point where two line segments meet. For instance, the ReLU function $\text{relu}(x)$ is not differentiable at $x = 0$, since the left derivative at $x = 0$ is 0, and the right derivative at $x = 0$ is 1. The same applies to our linearised functions. However, ReLUs are widely used, and are differentiated by arbitrarily choosing the derivative at $x = 0$ as either 0 or 1. Therefore, we have two options for training our networks:

1. We train our network with non-linear activation functions, but verify it and run it with piecewise-linear approximations.
2. We train our network with piecewise-linear approximations.

The first option has the advantage that we train with smooth, continuously differentiable activation functions. However, we train and verify with different architectures. As long as we verify and run the same object, this is not a problem for safety. It does raise a question: what is the effect of running a model trained with non-linear functions on a linearised architecture?

In Figure 8, we present the loss in test accuracy, as a result of transferring weights trained with the precise tanh function to networks with piecewise-linear approximations. If the tanh function is approximated with at least 3 line segments, the drop in accuracy is marginal.

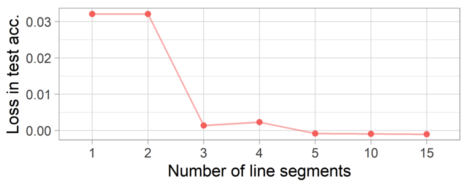


*Fig. 8: Loss from weight transfer (*tanh*).*

The second option has the advantage that we train and verify with the same architecture. Therefore, we do not incur the drop in accuracy which we expect

Hidden activation	Output activation	Training accuracy	Test accuracy	Training time (sec.)
Fully-connected network trained on MNIST (with PCA 25)				
relu	softmax	0.973	0.968	7.6
tanh	softmax	0.968	0.963	7.7
linear tanh	linear softmax	0.964	0.960	18.1
Convolutional network trained on MNIST				
relu	softmax	0.999	0.991	50.7
linear tanh	linear softmax	0.993	0.985	106.8
Convolutional network trained on CIFAR-10				
tanh	softmax	0.811	0.704	115.6
relu	softmax	0.925	0.782	115.1
linear tanh	linear softmax	0.769	0.702	243.9

Fig. 9: Performance for two networks trained on MNIST and one on CIFAR-10. For the linearised hidden activations, we use 3 segments. For the exp*-function in piecewise-linear softmax, we use 10 segments. We extrapolate the first and last line segments.*

from option (1). However, it does raise a different question: what is the effect of training with linearised activation functions, which have non-smooth gradients? We train a fully-connected and a convolutional neural network on the MNIST dataset and a convolutional neural network on the CIFAR-10 dataset [14]. Each architecture is trained with either the precise tanh and Softmax functions, or with their piecewise-linear approximations ($n = 5$). Since we did not observe any difference with respect to the different bounding methods, we only report the result for the extrapolation method. In Table 9, we show the results for these experiments. The drop in train and test accuracy of the fully-connected neural network trained and tested with linearised activation functions is marginal. For comparison we also train a convolutional neural network, and we observe that this model with linearised activations functions performs only slightly worse than a state-of-the-art model with ReLU activations.

5 Lessons Learned

Refinement types for neural network verification StarChild and Lazuli are flexible and lightweight libraries. They support the dimension-safe construction of neural networks. They support the lightweight verification of neural networks, in which neural networks and their verification conditions be written in the same language. Finally, they provide us with a user-friendly interface to SMT solvers, which means that merely *stating* the verification conditions is enough—the host language does the verification as part of type checking.

Training and verification in the same language We hope to extend our libraries with the ability to *train* as well as verify networks. However, there are several barriers to this. For F*, the main barrier is that code cannot be executed, but instead must be extracted to OCaml or F#. For Haskell, there already exist several Haskell-bindings for TensorFlow. However, at the time of writing, Liquid Haskell only verifies Haskell source, and not runtime objects such as neural network models. Hence, we would have to either extend Liquid Haskell with the ability to verify runtime objects, or convert the trained models to Haskell code. The former would constitute a significant contribution to Liquid Haskell, and the latter, while much simpler to implement, has very few advantages over our current approach.

Training networks using Keras made our work significantly easier, and importing the models to our libraries was an easy task. There is already existing work importing pre-trained models to theorem provers for the purposes of verification, *e.g.*, MLCert in Coq [4]. Our approach to importing models differs from MLCert: we translate floating-point numbers to F* reals, whereas MLCert translates them to bit-vectors.

Whether or not we integrate training into our libraries in the future, we believe that interfacing with the Python machine learning ecosystem will remain important for the foreseeable future.

Linearisation The method presented for scalable automatic linearisation in Section 4 works remarkably well. Our experiments show that it is possible to use piecewise-linear approximations of non-linear functions both during training and at runtime without a serious loss in accuracy. This is important, as non-linear real arithmetic with exponentials, logarithms, and trigonometric functions is undecidable, and therefore, it is unlikely that any future SMT solver will be able to efficiently decide problems of this sort.

Our current method of linearisation is crude, in that it splits the interval into sub-intervals of equal length. Often, a much better approximation is possible by varying the lengths of the sub-intervals.

Scalability and size reduction F* and Liquid Haskell offer to translate any program to the SMT logic. Unfortunately, this generality comes with a cost. In Figure 10, we present a benchmark for the verification of the n-ary AND gate, *i.e.*, the network which returns 1 if, and only if, each of its n inputs is 1. The verification task is to check whether the network returns the correct answer for each of four sample inputs. There are two curves for StarChild. One in which we use `assert`, and one in which we use `assert_norm`. Both are exponential. On the contrary, the line for Z3 does not exceed $1s$. Hence, it seems F* introduces an exponential factor in its encoding.

Unfortunately, while the curve for Z3 is encouraging, it does not scale to more complex conditions, such as the robustness conditions discussed in Section 3. Most solvers for linear real arithmetic simply do not scale to the size and complexity needed to check robustness conditions for even modest neural networks. There are several existing lines of work which attempt to address this problem. Marabou [13] uses a modification of the Simplex algorithm which more efficiently decides problems with piecewise-linear functions (such as ReLU). DeepPoly [23] uses abstract interpretations. Kwiatkowska [15] gives an overview of the progress in this area.

However, we consider the problem of scalable verification somewhat orthogonal to our goals. We seek to integrate existing solvers with programming languages in ways which make neural network verification as lightweight as possible. We used Z3 and other SMTLIB2-compatible solvers because these solvers have existing integration with programming languages. For future work, we plan to look into integrating Marabou with a dependently-typed programming language, and abandon generality in favour of generating efficient queries specific to the neural network domain.

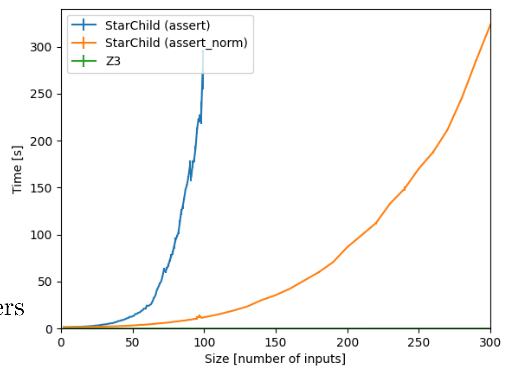


Fig. 10: Verification time for n-ary AND.

Soundness of the proposed approach We did not prove, or attempt to prove, that neural network transformations, such as size reduction (Section 3) or linearisation (Section 4) preserves the semantics of the network. Our assumption is that the verified network is deployed in practice, and we do not extend safety guarantees to the full precision network.

Whether or not this approach is practically feasible deserves further attention. There are multiple papers in the machine learning community that show that reduced size networks are feasible, and even desirable. There are a rising number of implementations of neural networks on special purpose hardware, *e.g.*, using FPGAs [26]), mobile phones [1], and special-purpose robotics hardware that require compression techniques. Ensuring that reduced-size networks perform sufficiently similar to the original networks is an optimisation problem that has been considered in the literature, and is beyond the limits of this study. However, we do provide a more detailed discussion of effects of linearisation in Section 4, as it is less well-studied in the literature.

Continuous training and verification In Section 1, we discussed why lightweight verification is appropriate for neural network verification. However, there is one novel feature of neural network verification, as opposed to the verification of conventional programs. Usually, we assume that the object we verify is uniquely defined, often hand-written, and therefore needs to be verified as-is. Neural networks are different—often there is a continuum of models that can serve as suitable classifiers. Given the task of verifying a neural network, we are no longer required to think of the object as immutable. This opens up new possibilities, where we can feed information from the verification process back into the training process. In fact, some papers in machine learning have already started to explore this fact [8].

Seen from this angle, methods such as dimensionality reduction and linearisation do not pose a threat to the soundness of our verification methods, but instead are a part of the conversation between the training and the verification mechanism in the search for a suitable, safe classifier.

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mane, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viegas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous distributed systems (2016)
2. Akbarpour, B., Paulson, L.C.: MetiTarski: An automatic theorem prover for real-valued special functions. Journal of Automated Reasoning **44**(3), 175–205 (Aug 2009)
3. Ayers, E.W., Eiras, F., Hawasly, M., Whiteside, I.: Parot: A practical framework for robust deep neural network training. In: Lee, R., Jha, S., Mavridou, A. (eds.) 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)
4. Bagnall, A., Stewart, G.: Certifying true error: Machine learning in Coq with verified generalisation guarantees. AAAI (2019)
5. Barrett, C., Stump, A., Tinelli, C., et al.: The smt-lib standard: Version 2.0. In: Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England). vol. 13, p. 14 (2010)
6. Chollet, F., et al.: Keras. https://keras.io (2015)
7. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(t). In: Computer Aided Verification, pp. 81–94. Springer Berlin Heidelberg (2006)
8. Fischer, M., Balunovic, M., Drachsler-Cohen, D., Gehr, T., Zhang, C., Vechev, M.T.: DL2: training and querying neural networks with logic. In: Proc. of the 36th Int. Conf. Machine Learning, ICML 2019. vol. 97, pp. 1931–1941. PMLR (2019)
9. Fisher, K., Launchbury, J., Richards, R.: The HACMS program: using formal methods to eliminate exploitable bugs. Phil. Trans. Royal Society (2017)
10. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572 (2014)
11. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: CAV 2017. vol. LNCS 10426, pp. 3–29. Springer (2017)
12. Jovanović, D., de Moura, L.: Solving non-linear arithmetic. ACM Communications in Computer Algebra **46**(3/4), 104 (Jan 2013)
13. 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)
14. Krizhevsky, A., Hinton, G., et al.: Learning multiple layers of features from tiny images. Tech. rep., Citeseer (2009)

15. Kwiatkowska, M.Z.: Safety verification for deep neural networks with provable guarantees (invited paper). In: Fokkink, W., van Glabbeek, R. (eds.) CONCUR 2019,. LIPIcs, vol. 140, pp. 1:1–1:5. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)
16. LeCun, Y., Cortes, C., Burges, C.: Mnist handwritten digit database. ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist **2** (2010)
17. McCulloch, W., Pitts, W.: A logical calculus of the ideas immanent in nervous activity. Bulletin of Math. Bio. **5**, 115–133 (1943)
18. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS'08. LNCS, vol. 4963, pp. 337–340 (2008)
19. Papernot, N., McDaniel, P.D., Swami, A., Harang, R.E.: Crafting adversarial input sequences for recurrent neural networks. In: Brand, J., Valenti, M.C., Akinpelu, A., Doshi, B.T., Gorsic, B.L. (eds.) 2016 IEEE Military Communications Conference, MILCOM 2016, Baltimore, MD, USA, November 1-3, 2016. pp. 49–54. IEEE (2016)
20. Parisi, G., et al.: Continual lifelong learning with neural networks: A review. Neural Networks **113**, 54 – 71 (2019)
21. Pertigkiozoglou, S., Maragos, P.: Detecting adversarial examples in convolutional neural networks. CoRR **abs/1812.03303** (2018), http://arxiv.org/abs/1812.03303
22. Serban, A.C., Poll, E.: Adversarial examples - A complete characterisation of the phenomenon. CoRR **abs/1810.01185** (2018), http://arxiv.org/abs/1810.01185
23. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: An abstract domain for certifying neural networks. PACMPL **3**(POPL), 41:1–41:30 (2019)
24. Swamy, N., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y.: Dependent types and multi-monadic effects in F*. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2016. ACM Press (2016)
25. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., Fergus, R.: Intriguing properties of neural networks. arXiv preprint arXiv:1312.6199 (2013)
26. Umuroglu, Y., Fraser, N.J., Gambardella, G., Blott, M., Leong, P.H.W., Jahre, M., Vissers, K.A.: FINN: A framework for fast, scalable binarized neural network inference. In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017, Monterey, CA, USA, February 22-24, 2017. pp. 65–74 (2017)
27. Vazou, N.: Liquid Haskell: Haskell as a Theorem Prover. Ph.D. thesis, University of California, San Diego, USA (2016)
28. Vazou, N., Tondwalkar, A., Choudhury, V., Scott, R.G., Newton, R.R., Wadler, P., Jhala, R.: Refinement reflection: complete verification with SMT. Proceedings of the ACM on Programming Languages **2**(POPL), 1–31 (Jan 2018)