Verifying Feedforward Neural Networks for Classification in Isabelle/HOL*

Achim D. Brucker and Amy Stell

Department of Computer Science
University of Exeter
Exeter, UK
{a.brucker,as1343}@exeter.ac.uk

Abstract. Neural networks are being used successfully to solve classification problems, e.g., for detecting objects in images. It is well known that neural networks are susceptible if small changes applied to their input result in misclassification. Situations in which such a slight input change, often hardly noticeable by a human expert, results in a misclassification are called adversarial examples. If such inputs are used for adversarial attacks, they can be life-threatening if, for example, they occur in image classification systems used in autonomous cars or medical diagnosis.
Systems employing neural networks, e.g., for safety or security-critical functionality, are a particular challenge for formal verification, which usually expects a formal specification (e.g., given as source code in a programming language for which a formal semantics exists). Such a formal specification does, per se, not exist for neural networks.
In this paper, we address this challenge by presenting a formal embedding of feedforward neural networks into Isabelle/HOL and discussing desirable properties for neural networks in critical applications. Our Isabelle-based prototype can import neural networks trained in TensorFlow, and we demonstrate our approach using a neural network trained for the classification of digits on a dot-matrix display.

Keywords: Neural network · Deep Learning · Classification Network · Feedforward Network · Verification · Isabelle/HOL

1 Introduction

Deep learning, i.e., machine learning using neural networks is used successfully in many application areas, e.g., image classification ([24, 37, 11]). While systems using neural networks are also used in safety-critical areas (e.g., for the recognition of street signs in semi-autonomous cars [11]), the use of neural networks in high-assurance systems is limited due to the lack of formal verification techniques that satisfy the stringent requirements of industrial certification standards

* This work was supported by the Engineering and Physical Sciences Research Council [grant number 670002170].

M. Checkik, J.-P. Katoen, and M. Leucker (Eds.): FM 2023, LNCS, pp. 1–18, 2023.
© 2023 Springer-Verlag. This is the author's version of the work. It is posted at https://www.brucker.ch/bibliography/abstract/brucker.ea-feedforward-nn-verification-2023 by permission of Springer-Verlag for your personal use. BibTeX, Word, EndNote, RIS

such as BS EN 50128 [10] (safety) or Common Criteria [14] (security) that are required for such applications.

The formal specification and verification techniques that such standards require usually rely on the existence of an implementation (e.g., source code) whose compliance to a specification can be verified (e.g., following an approach similar to [23]). For systems based on neural networks, such an implementation that precisely describes, in a human-readable form, the system's behaviour does not exist. The only artefact that exists is a neural network trained on a set of training data, which is expected to behave correctly for all possible inputs.

Formal verification is an approach that can make a statement for all possible inputs. In this paper, we present an approach based on the interactive theorem prover Isabelle/HOL for the formal verification of neural networks. Using an expressive formalism, such as higher-order logic, allows for expressing complex properties that a neural network needs to satisfy. On the one hand, the fact that Isabelle is an interactive theorem prover enables the user to explore the properties of the network and, therefore, deepen the understanding of the neural network being analysed. On the other hand, Isabelle is a framework that allows us to provide highly automated functionality for both, encoding a specific neural network, and for proving properties over it.

In more detail, our contributions are three-fold:

1. a formal embedding of feedforward neural networks into Isabelle/HOL,
2. a verification environment supporting the verification of neural networks trained using TensorFlow, and
3. an application of our framework to a case study.

The rest of the paper is structured as follows: first, we introduce the background of our work (Sect. 2) and a small running example (Sect. 3). Next, we introduce our formal model of feedforward neural networks in Isabelle/HOL in Sect. 4 and discuss the desirable properties of classification networks in Sect. 5. In Sect. 6, we briefly explain our implementation in Isabelle/HOL before we briefly discuss a case study for classifying dot-matrix digits (Sect. 7). Finally, we discuss related work (Sect. 8) and draw conclusions (Sect. 9).

2 Isabelle and Higher-Order Logic (HOL)

In this section, we introduce two aspects of Isabelle/HOL; its logic (HOL) and its implementation architecture.

2.1 Isabelle/HOL

Isabelle [28] is a well-known interactive theorem prover that has been used successfully in large-scale verification projects (e.g., [23] presents the verification of an operating system kernel using Isabelle/HOL). The formal language of Isabelle is *Higher-order logic* (HOL) [12], which is a classical logic based on a simple type system. It provides the usual logical connectives like $\neg$ `_`, `_` $\wedge$ `_`, `_` $\vee$ `_`, `_` $\longrightarrow$ `_` as well as the object-logical quantifiers $\forall$ `x. P x` and $\exists$ `x. P x`. In contrast

to first-order logic, quantifiers may range over arbitrary types, including total functions `f::`$\alpha \Rightarrow \beta$ (where α and β are polymorphic type variables).

Isabelle/HOL offers support for extending theories in a logically safe way: A theory-extension is *conservative* if the extended theory is consistent, provided that the original theory was consistent. Conservative extensions can be *constant definitions*, *type definitions*, *datatype definitions*, *primitive recursive definitions* and *well-founded recursive definitions*.

2.2 Isabelle as Formal Methods Framework

Isabelle is not only an interactive theorem prover; it also provides an extensible framework for developing formal method tools [39]. Fig. 1 shows an overview of the Isabelle architecture. For our work, it is noteworthy that new components can

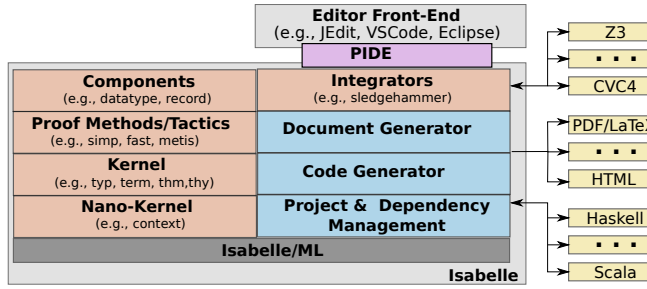


Fig. 1: The system architecture of Isabelle.

be implemented in Isabelle/ML, i.e., Isabelle's SML [29] programming interface. In a logically safe way, we use this interface to provide an import mechanism for importing neural networks and implementing domain-specific proof methods. Furthermore, use the code generator to efficiently evaluate neural networks, i.e., compute predictions for concrete inputs.

3 Running Example: Classifying Lines in a Grid

In the following, we introduce neural networks for (image) classification by using a simple line classification problem: given a 2×2 pixel greyscale image, the neural network should decide if the image contains a horizontal line (e.g., Fig. 2a), vertical line (e.g., Fig. 2b), or no line (Fig. 2c).

Traditionally, textbooks (e.g., [3]) define a feedforward neural network as directed weighted acyclic graphs. The nodes are called *neurons*, and the incoming edges are called *inputs*. For a given neuron k with m inputs x_{k_0} to $x_{k_{m-1}}$, and

(a) horizontal line (b) vertical line (c) no line (d) misclassification

Fig. 2: Example input images to our classification problem.

the respective weights w_{k_0} to $w_{k_{m-1}}$ the neuron computes the output

$$y_k = \varphi\left(\beta + \sum_{j=0}^{m} w_{k_j} x_{k_j}\right) \tag{1}$$

where φ is the *activation function* and β the *bias* for the neuron k. The values for the weights and biases are determined during the training (learning) phase, which we omit due to space reasons. In our work, we assume that the given neural network is already trained, e.g., using the widely used machine learning framework TensorFlow [1].

Fig. 3 illustrates the architecture of our neural network: The neural network for our example classification problem has four inputs (one for each pixel of the image), expecting an input value between 0.0 (white) and 1.0 (black). It also

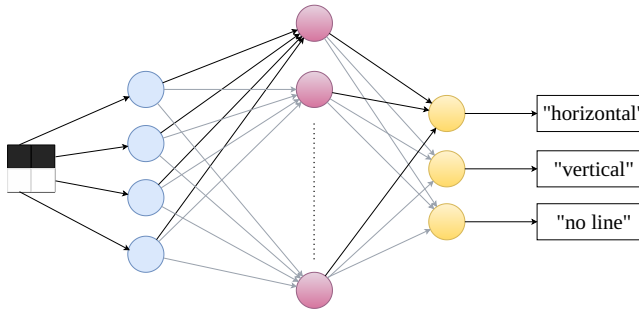


Fig. 3: Neural network for classifying lines in 2×2 pixel greyscale images.

has three outputs, one for each possible class (a horizontal line, a vertical line, or no line). The neurons (nodes) can be naturally categorised into layers, i.e., the *input layer* consisting out of the input nodes and the *output layer* consisting out of the output nodes. Moreover, our neural network has one *hidden layer* with 16 neurons. The input layer and the hidden layer use a linear activation function (i.e., $\varphi(x) = x$) for all neurons, and the hidden layer uses the binary step function (i.e., $\varphi(x) = 0$ for $x \leq 0$ and $\varphi(x) = 1$ otherwise). In our example, there is an edge between each neuron from the previous layer to the next layer, often called a *dense layer*. Machine learning approaches using neural networks with one or more hidden layers are called *deep learning*.

In our example, we used the Python API for TensorFlow [1] to train our neural network. We obtained a neural network that reliably classifies black lines in a given 2×2 image with 100% accuracy. While this sounds great, the neural network is not very resilient to changes to its input values. Consider, for example, Fig. 2d: a human expert would likely classify this image as "no line". Nevertheless, our neural network classifies this as a horizontal line, even though the upper right pixel is only light grey with a numerical value of 0.05, much closer to white than black. Such a misclassification is usually called an *adversarial example*. If such a network is used in safety or security-critical applications, e.g., for classifying street signs, such misclassifications can be life-threatening.

4 Modelling Neural Networks in Isabelle

Our Isabelle/HOL formalisation contains several models, i.e., one based on modelling neural networks as graphs (i.e., "textbook-style") and one modelling neural networks as layers (i.e., "TensorFlow-style"). Due to space reasons, we will focus in this paper, on the latter.

4.1 Data Modelling

We use locales (i.e., Isabelle's mechanism for parametric theories) to capture fundamental concepts that are shared between different models of neural networks. We start by defining a locale `neural_network_sequential_layers` to describe the common concepts of all neural network models that use layers are core building blocks. For our representation to be a well-formed sequential model, we require that the first layer is an input layer and the last layer is an output layer:

Isabelle (Isar)
```
locale neural_network_sequential_layers =
  fixes N::⟨('a::{monoid_add,times}, 'b) neural_network_seq_layers⟩
  assumes head_is_In:  ⟨isIn (hd (layers N))⟩
  and     last_is_Out: ⟨isOut (last (layers N))⟩
  and     ⟨list_all isInternal  ((tl o butlast) (layers N))⟩
begin end
```

For this encoding of a neural network, we mostly follow TensorFlow's Sequential model [1] and represent our network as a list of layers with an abstract table of activation functions, allowing for extensible and customisable functionality. The record `('a, 'b) neural_network_seq_layers` represents our network where `'a` is an abstract value representing the type of our weights and bias, and `'b` is our activation function.

Isabelle (Isar)
```
record ('a, 'b) neural_network_seq_layers =
    layers ::  ⟨('a, 'b) layer  list⟩
    activation_tab :: ⟨'b ⇒ (('a list ⇒ 'a list) option)⟩
```

Included in our formalisation are definitions for all TensorFlow [1] activation functions, and for those which use e^x, we also provide an approximation using the Taylor series of the exponential function, which has been shown to outperform the original in certain situations [4]. In our running example (recall Sect. 3), the activation functions used during training include binary step in the hidden layer and linear in the output layer.

Isabelle (Isar)
```
definition
  ⟨identity = (λv. v)⟩
definition binary_step :: ⟨'a::{zero, ord, one, zero} ⇒ 'a⟩ where
 ⟨binary_step = (λ v. if v ≤ 0 then 0 else 1)⟩
```

As we are using a representation of a network as a list of layers, we also support different layer types and their computations. Currently, our sequential layers model supports five layer types *Input*, *Output*, *Dense*, *Activation*, and, as we allow for the abstraction of activation functions, we can define arbitrary `'b` in the networks `activation_tab`, allowing for custom activation functions. Therefore, we do not need to model TensorFlow's *Lambda* layer explicitly (which is TensorFlow's mechanism for supporting custom activation functions).

Isabelle (Isar)
```
datatype ('a, 'b) layer = In ⟨InOutRecord⟩
                        | Out ⟨InOutRecord⟩
                        | Dense ⟨('a,'b) LayerRecord⟩
                        | Activation ⟨('b) ActivationRecord⟩
```

These layer types differ in how they are connected to the next layer in the network, thus changing the calculation during training and prediction. The Dense layer is the most powerful layer type in the sense that it connects all outputs of the previous layer with all inputs. Hence, other layer types (e.g., TensorFlow's *Activation* layer, which applies an activation function to each output of the previous layer) can be expressed in terms of a Dense layer with certain weights set to the constant 0 to "disable" certain edges.

Each `('a, 'b) LayerRecord` contains the activation, weights and bias in our network (φ, β and ω respectively), while our `('b) ActivationRecord` only contains our abstracted activation function.

Isabelle (Isar)
```
record InOutRecord =
       name:: String.literal
       units:: nat
record ('b) ActivationRecord = InOutRecord +
      φ :: 'b
record ('a, 'b) LayerRecord = ⟨('b) ActivationRecord⟩ +
      β :: ⟨'a list⟩
      ω :: ⟨'a list list⟩
```

4.2 Encoding our Running Example

Using the above definitions, we can now show the specialisation of our formalisation by explaining the representation of our network discussed in Sect. 3, in Isabelle/HOL. We represent this example by first defining the types of our concrete network, as the encoding of the grid uses an array of NumPy [19] 64-bit floats, the `'a` in our record `('a, 'b) neural_network_seq_layers` is instantiated as a real and the `'b`, is of the datatype $\texttt{activation}_{\texttt{multi}}$, (a datatype that allows for the mapping of the abstraction of multi-class activation functions onto its Isabelle/HOL definition).

Next, we have the layers; the input layer is a densely connected layer that passes each input into each neuron in the first hidden layer.

Isabelle (Isar)
```
dense_input ≡ (| name = STR ''dense_input'', units = 4 |)
```

The hidden layer in the network is a dense layer with 16 units, the learned weights and bias referenced in this layer refer to the connections that exist between this and the previous input layer.

Isabelle (Isar)
```
dense ≡ (| name = STR ''dense'', units = 16,
           φ = mBinaryStep, β = [5 / 10, ..., - 145 / 10],
           ω = [[1, ..., 1] ..., [8, ..., 8]]|)
```

The next layer is the final calculation layer in our network and passes the results onto our final output layer, which outputs the prediction of the network.

Isabelle (Isar)
```
dense_1 ≡ (| name = STR ''dense_1'', units = 3,
             φ = mIdentity, β = [1, 0, 0],
             ω = [[0, 0, 0], ...,[0, 0, 0]]|)
OUTPUT ≡ (| name = STR ''OUTPUT'', units = 3|)
```

Using the above layer and the activation function definitions; our final neural network for the classification of horizontal and vertical lines can be defined as follows:

Isabelle (Isar)
```
NeuralNet ≡ (| layers = [dense_input, Layers.dense,
               dense_1, OUTPUT], activation_tab = grid.φ_grid|)
```

4.3 Evaluating Neural Networks

What remains is the evaluation of the network, usually called "prediction". To be able to verify that a network's behaviour falls within our desirable properties (Sect. 5), we need to be able to efficiently evaluate its prediction for a given input. As the calculation performed depends on the layer of the network that we are currently evaluating, we calculate the output based on the layer type and fold this over the network.

The input and output layers of our network pass the inputs directly onto the next layer without any calculation performed; this can be seen in the first two cases of the $\texttt{predict}_{\texttt{layer}}$ function. The dense layer of the network is where Equation 1 is calculated, case three in $\texttt{predict}_{\texttt{layer}}$, where first the input weights are transposed (`in_weights`), then zipped with their input value (`in_w_pairs`), before calculating the weighted sum (`wsums`), adding the bias (`wsum_bias`), and finally applying the activation function on the result, producing the output for a single dense layer. To calculate the prediction of the network given a set of inputs we then fold $\texttt{predict}_{\texttt{layer}}$ over the network from left to right (`foldl`) in $\texttt{predict}_{\texttt{seq\_layer}}$

It is within this function that we also specify some pre-conditions for the network to be of a valid structure. For example, the length of the input vector must be equal to the number of units in that layer (`length vs = l`), for the activation, input, and output layers; if this is not the case, then we return the `None` option type, indicating that an error has occurred in prediction.

Isabelle (Isar)

```
fun predict_layer::⟨('a, 'b) neural_network_seq_layers
    ⇒ ('a list) option ⇒ ('a, 'b) layer ⇒ ('a list) option⟩ where
  ⟨predict_layer N (Some vs) (In ⦇name = _, units = l⦈)
              = (if length vs = l then Some vs else None)⟩
| ⟨predict_layer N (Some vs) (Out ⦇name = _, units = l⦈)
              = (if length vs = l then Some vs else None)⟩
| ⟨predict_layer N (Some vs) (Dense pl)  = (let
              in_weights = convert_weights (ω pl);
              in_w_pairs = map (λ e. zip vs e) in_weights;
              wsums      = map (λ vs'. ∑(x,y)←vs'. x*y) in_w_pairs;
              wsum_bias  = map (λ (s,b). s+b) (zip wsums (β pl))
              in (case activation_tab N (φ pl) of
                       None   ⇒ None
                     | Some f ⇒ Some (f wsum_bias)))⟩
| ⟨predict_layer N (Some vs) (Activation pl) =
       (if length vs = units pl then (case activation_tab N (φ pl) of
                                        None ⇒ None
                                      | Some f ⇒ Some (f vs))
                                else None)⟩
| ⟨predict_layer _ None _ = None⟩

definition
    ⟨predict_seq_layer N xs = foldl (predict_layer N) (Some xs) (layers N)⟩
```

Although this model of a neural network differs from the textbook definition of a network represented as a weighted and directed graph [3], this encoding follows closely that of TensorFlow [1] where their sequential model consists of an ordered list of layers, in which the activation is consistent within a single layer, and has added support for various layer types. As well as this, our sequential layers model resembles the original vector representation of Rumelhart et al. [32]. However, modelling a network as a list of layers means that it is not appropriate

for networks with multiple inputs and outputs, as well as those that have layer sharing and multiple branches. In order to model these networks, we have also developed a formalisation that utilises graph theory and encodes a network as a weighted and directed acyclic graph, allowing the specification of arbitrary connections between layers, including a non-linear topology, it is however, less computationally efficient.

4.4 Compliance of our Formalisation to TensorFlow

To ensure that our formalisation is a faithful representation of the neural networks that we defined in TensorFlow, we provide a framework that supports the import of trained TensorFlow networks and their test data. We can then use this to evaluate our Isabelle network and validate that the output is the same, hence providing confidence that our formalisation is accurate.

Similar to what we will discuss in Sect. 6, we can import text files containing NumPy [19] arrays of our test inputs, expectations and predictions from our trained TensorFlow network.

Isabelle (Isar)
```
import_data_file file defining inputs
```

We can now prove that our formally encoded neural network computes the same prediction (within an error interval) as TensorFlow. To express this requirement, we first define `check_result_list_interval` for checking that two lists are approximatively equal (we need the error interval due to possible rounding errors in IEEE754 arithmetic in python compared to mathematical reals in Isabelle).

Isabelle (Isar)
```
fun check_result_list_interval where
⟨check_result_list_interval None None           = True⟩
| ⟨check_result_list_interval (Some xs) (Some ys)
        = fold (∧) (map2 (λ x y. x ∈ set_of y)  xs ys) True⟩
| ⟨check_result_list_interval _ _ = False⟩
notation check_result_list_interval (((_)/ ≈₁ (_)) [60, 60] 60)
```

Using `check_result_list_interval`, we now define the property that the (symbolically) computed predictions of a neural network meet our expectations:

Isabelle (Isar)
```
definition
  ensure_testdata_interval :: ⟨real list list
      ⇒ (real list ⇒ real list option)
      ⇒ real interval list list ⇒ bool⟩ where
 ⟨ensure_testdata_interval inputs P outputs =  foldl (∧) True
   (map (λ e. let a = (P (fst e)) in let b = Some (snd e) in (a ≈₁ b))
          (zip inputs outputs))⟩
notation ensure_testdata_interval (⊢\ᵢ {(_)} (_) {(_)} [3, 90, 3] 60)
```

For our example, we can now prove the following lemma:

Isabelle (Isar)
```
lemma grid_meets_predictions:
⟨⊢ᵢ {inputs} (predict_seq_layer NeuralNet) {i_of 0.000001 predictions}⟩
by(simp add: ensure_testdata_interval_def upper_Interval lower_Interval
             predictions_def i_of_def inputs_def in_set_interval)
```

Where `i_of 0.000001 predictions` computes intervals with the expected predictions as midpoints, i.e., given an expectation p, our lemma shows that the actual prediction p' of our formal neural network is satisfies $|p - p'| \leq 0.000001$.

This lemma is proven by unfolding all definitions using Isabelle's simplifier, which corresponds to a symbolic execution of the prediction function. Hence, we can be sure that our formal model behaves identically to the model executed on TensorFlow on a concrete set of input data.

Many classification networks use the maximum output as the result, without normalisation (e.g., to values between 0 and 1). In such cases, a weaker form of ensuring compliance to predictions might be used that only checks that checks for the maximum output of each given input:

Isabelle (Isar)
```
definition
  ensure_td_max :: ⟨real list list ⇒ (real list ⇒ real list option)
      ⇒ real list list ⇒ bool⟩ where
  ⟨ensure_td_max inputs P outputs
   =  foldl (∧) True
      (map (λ e. case P (fst e) of
                   None ⇒ False
                 | Some p ⇒ map_option fst (pos_of_max p)
                             = map_option fst (pos_of_max (snd e)))
           (zip inputs outputs))⟩
  notation ensure_td_max (⊢ {(_)} (_) {(_)} [3, 90, 3] 60)
```

We will see an application of this check in Sect. 7.

5 Properties of Classification Networks

In contrast to traditional program verification, for neural networks, there has yet to be an established notion of safety or correctness of a trained neural network. Recently there has been more of a discussion in this area of different types of properties that should hold for arbitrary networks [33] (discussed in more detail in Sect. 8). However, for our example, we focus on looking at input-output relations and notions of robustness within neighbourhoods of the input.

For example, Pulina et al. [30] consider a network safe, if given every possible input, its output is guaranteed to range within specific bounds. This is motivated by an application in which, e.g., a neural network 'computes' dosages of a drug. In this case, there are certain maximums (or minimums) that are considered to be not safe. This is a property we can easily express in our framework as constraints of the range of computing predictions of a given network.

For pure classification networks, which is our focus in this paper, one is usually only interested in the maximum value of the classification outputs (and only to a lesser extent to its actual value). Often, classification networks use activation functions (such as softmax) that normalise the outputs, or argmax that only outputs the maximum classifier. For our running example, we can easily prove:

Isabelle (Isar)
```
lemma
⟨ran (predict_seq_layer NeuralNet xs) ⊆ {[0, 0, 1],[0, 1, 0],[1, 0, 0]}⟩
```

Where `ran` is the range operator of HOL. While not a safety property in the traditional sense, this lemma shows that the output of the classification is never ambiguous (i.e., two or more classification output having the value 1).

In a more generalised form, Kurd et al. [25] define safety as a clearly defined input-output-relation, i.e., satisfying a given function (or, in our notation, a higher-order predicate) that is tested on known and unknown inputs. Moreover, the behaviour should be repeatable and predictable, it should also tolerate faults in inputs (e.g., where inputs lie outside a specified set of inputs), and no hazardous outputs (e.g., no output outside the range of the target function) should be predicted. Very similar is the idea of Huang et al. [20], who define safety as the requirement that small changes to an input should not change the classification. For our running example, we can express such a verification goal as follows:

Isabelle (Isar)
```
lemma ⟨x3 ∈ {0.96..1.00} ∧ x2 ∈ {0.96..1.00}
     ∧ x1 ∈ {0.00..0.04} ∧ x0 ∈ {0.00..0.04}
     ⟹  predict_seq_layer NeuralNet [x3, x2, x1, x0] = Some [0, 1, 0]⟩
```

This lemma, which we have proven in Isabelle/HOL (including the corresponding lemmata for the other output classes of our example), states that the classification of the upper horizontal line is resilient to small changes in the greyscale values of the pixels (e.g., caused by dust turning white into a greyish colour or a very bright light that might turn black into a dark grey). While looks good "on paper", it is actually showing the opposite. Already small changes in the colour values that are unlikely to be detected by the naked eye, can result in a misclassification (recall Fig. 2d).

The last example also shows that we will need to develop domain-specific failure models (e.g., modelling the impact of non-optimal camera angles or light conditions), which can then form the basis for deriving safety properties for applications that rely on neural networks. Broadly speaking, this is also suggested by Katz et al. [21] that, in a case study for aircraft avoidance detection, use a notation of unnecessary turning advisories to show that the trained neural network does not omit them.

In addition, there are further properties that we formalised and that can increase the confidence in the predictions of a neural network by reducing the likelihood of ambiguous classification results. This includes, e.g., the requirement that for a given input, the classification outputs have at least a given minimum

distance (e.g., avoiding situations where all classification outputs show nearly identical values):

```
definition                                                    Isabelle (Isar)
ensure_delta_min :: ‹real ⇒ (real list ⇒ real list option) ⇒ bool›
where ‹ensure_delta_min δ P = (∀ xs ∈ ran P. δ ≤ δ_min xs)›
notation ensure_delta_min ((_) ⊢ (_) [61, 90] 60)
```

6 Implementation

We implemented our approach in Isabelle/HOL, i.e., we made use of the ability of Isabelle to extend it with new datatype packages and proofs (see Fig. 4 for an overview of the extended architecture). In particular, we developed a datatype

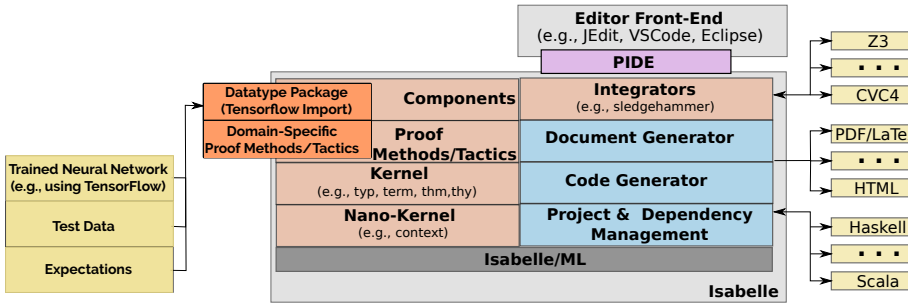


Fig. 4: The system architecture of our architecture, adding a datatype package and custom proof methods to Isabelle/HOL.

package that can import trained neural networks using the JSON [16]-based format used by TensorFlowJS [35]:

```
import_TensorFlow grid file model.json as seq_layer          Isabelle (Isar)
```

Our new Isabelle/Isar [40] command `import_TensorFlow` encodes the neural network model stored in the file `model.json`[1] as sequence of layers (`seq_layer`), i.e., the formal encoding described in Sect. 4 (our datatype package also supports alternative formal encoding, e.g., one that models neural networks as directed graphs). Our datatype package also proves that the imported model complies with the requirements of our formal model (technically, this is done by an automated instantiation proof for the locale `neural_network_sequential_layers`)

[1] TensorFlowJS stores the structure of the machine learning model in a JSON [16]-based format that refers to a binary file containing the weights and biases. Our import mechanism fully supports this format, i.e., also importing the weights and biases from the external file.

as well as proves various auxiliary properties (e.g., conversion between different representations) that can be useful during interactive verification.

Our datatype package also supports the automatic import of test data or prediction computed by, e.g., TensorFlow [1], using the data format of NumPy [19]:

```
import_data_file predictions.txt defining predictions
```
Isabelle (Isar)

This command imports the prediction data, i.e., input data and expected outputs, into Isabelle and binds it to the logical constant `predictions`. This data can later be used in a formal proof that the imported model has a certain accuracy on this data set.

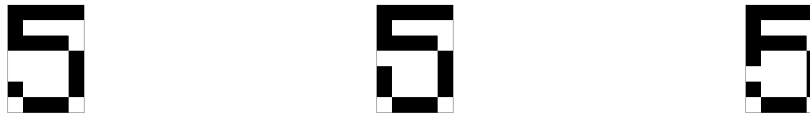
Finally, we started to develop domain-specific proof tactics or methods using Eisbach [26], e.g., for the selective unfolding of generated definitions or providing optimised configurations for the symbolic evaluation of the prediction function for neural networks.

Overall, our prototype enables a workflow in which one trains a neural network using, e.g., the Python API for TensorFlow and exports the model and the training and prediction data. This data can then be used to prove that the formal model is semantically equivalent to the original model. Alongside this, we also have an external tool that can convert networks saved in ONNX (`https://onnx.ai/`) format, providing they have an architecture that our formalisation supports, into the JSON representation we currently require.

Our formalisation comprises over 5300 lines of formal definitions and generic proofs in Isabelle/HOL. The implementation of the datatype package adds another 1000 lines of Isabelle/ML code (not including the JSON-parser and the basic datatype package for JSON, both provided by [8]).

7 Classifying Digits of a 5 × 7 Matrix Display

In this section, we briefly discuss a larger case of a neural network for the classification of digits on a dot-matrix display (see Fig. 5a). As for our running example, we used the Python API of TensorFlow [1] for training the network.



(a) (b) (c)

Fig. 5: The Digit 5 on a 5 × 7 Matrix Display.

Our network has 35 (= 5 · 7) neurons in the input layer and 10 neurons in the output layer. While our running example (recall Sect. 3) ensures that the

output values are between 0 and 1, our neural network for the digit classification has a "non-normalised" output, performing a maximum classification.

Consequently, to convince ourselves that the formal representation of our classification network complies with the TensorFlow representation, we prove:

Isabelle (Isar)
```
lemma digits_meets_expectations_max_classifier:
      ⟨⊢ {inputs} (predict_seq_layer digits.NeuralNet) {expectations}⟩
```

Where `digit.NeuralNet` is the formal representation of our neural network, `inputs` is a list of input values and `expectations` the corresponding expectations (classification). Recalling Sect. 5, we note that this lemma uses the higher-order predicate `ensure_td_max` (`⊢ {_} _ {_}`), which does not require that the predictions lie within a specific interval. Instead, it requires that the maximum classifier of the actual and expected predictions are the same.

Furthermore, we show that an arbitrary one-pixel failure (e.g., a dead pixel or, a pixel that constantly is switched on, or any value in-between) does not change the classification. This is formally expressed as follows:

Isabelle (Isar)
```
lemma assumes xs: ⟨xs = [x34,x33, ... , x0]⟩
and limits: ⟨set xs ⊆ {0..1}⟩ (* grey scale pixels *)
and h: ⟨hamming (digits!5) xs ≤ 1⟩
shows ⟨classify_as xs 5⟩
```

Here we make use of an auxiliary predicate for capturing the fact that the network did classify the input as a certain digit:

Isabelle (Isar)
```
definition classify_as::⟨real list ⇒ nat ⇒ bool⟩ where
⟨classify_as xs n = (Option.bind (predict_seq_layer digits.NeuralNet xs)
                          pos_of_max = Some n)⟩
```

We model 1-pixel changes by requiring that the Hamming distance representation of the digit 5 (`digits!5`) is at most one. Thus, we have a formal proof that our neural network classifies any image that deviates from an ideal five only by one pixel, reliably as five. Consider, for example, Fig. 5c for which a human could already be uncertain if the image shows a five or a six, even though only one pixel has been changed.

8 Related Work

Using Isabelle/HOL for AI verification. To the best of our knowledge, there are no examples of formalising neural networks in an Interactive Theorem Prover, and very few examples of formalising machine learning. In Isabelle/HOL the closest related work is by Bentkamp et al [6] which formalises the expressiveness of deep learning. Based on the theoretical work by Cohen et al. [13], verifies the superiority of deep learning over shallow learning. Abdulaziz et al [2] have

formally verified the AI planning problem using a SAT encoder, with the formalisation showing that the SAT-based planner Madagascar [31] falsely claims that problems have no solutions of certain lengths.

Neural network verification. Many traditional approaches to formal methods and safety verification are insufficient in the case of neural networks as there is no complete specification for their behaviour. Approaches are generally categorised into exact verifiers and incomplete but more efficient verification techniques. On the latter, which solves a relaxed problem that is more computationally efficient, methods include abstract interpretation [27], linear relaxations[38] and duality [15]. Using abstraction, infinite behaviours can be approximated using a finite representation by abstract transformers that are used to capture approximations of network layer computations. The problem then becomes reducing these over-approximations to more precisely capture the behaviour without introducing computational complexity. Most examples still work on ReLU networks [34], however, there has been some recent progress in developing abstract transformers for softmax and other difficult activations [7].

Among the complete verification techniques, most utilise SAT/SMT solvers, or Mixed Integer Linear Programming (MILP). Approaches include those that take advantage of piecewise linear activation functions, which are more manageable for network verification. Work includes that by Szegedy et al [36], who ensure that networks assign correct scores to the output advisories in various input domains. Planet [17], which presents an approach using SAT solving combined with linear programming to cut out significant areas of the search space during verification. Similarly, Reluplex, [21], is able to prove many robustness properties of larger-scale neural networks with ReLU activation functions and has recently been expanded into Marabou, [22], for arbitrary piecewise linear activation functions.

While these approaches lead to shorter verification times, the problem of complete verification of non-linear activation functions remains limited to smaller networks, and while approaches using approximation methods that allow for these activation functions are sound, they remain incomplete. By using an interactive theorem prover, as opposed to SMT/SAT solvers, we are able to make use of higher-order logic to define, reason over, and verify our new datatypes and definitions by building on mathematical axioms, whilst still maintaining flexibility and efficiency.

Properties. While the properties we discuss and verify are mostly concerned with the input-output relations, there is a general lack of a widely accepted formal specification when concerned with neural networks. Most frequently, the desired behaviours discussed include predictability of the behaviour and tolerance to faulty input [25], looking at improving the stability of the classification [20], or a general idea of robustness around a specific input region, where manipulations applied does not cause misclassifications. However, more recently, other properties have been discussed, such as semantic invariance [18], fairness [5], or distributional assumptions [33]. While these are all relevant and important

properties for a network to fulfil, they currently have a less precise formal specification and so currently have limited application in formal methods, yet are interesting avenues for future research.

9 Conclusion and Future Work

We presented a formalisation of feedforward neural networks in Isabelle/HOL. To the best of our knowledge, this is the first formalisation of neural networks in an interactive theorem prover. We also made use of the framework aspect of Isabelle to provide an import mechanism automating our encoding for neural networks stored in a widely used exchange format.

Still, we see our work only as the beginning of a journey towards formally verified safety and correctness guarantees for critical systems employing ML/AI-based components. On a general level, there is further work required to improve the understanding of what it means that a neural network is safe (and secure), and how to convert this into a formal specification. This discourse will, hopefully, result in further properties that can be used in formal verification, and that will allow a comparison amongst various formal approaches for the verification of neural networks.

More specific to our approach, we plan to extend the types of layers and architectures that are directly supported, which, together with developing domain-specific proof tactics, should increase the degree of proof automation significantly. Modelling additional representations (e.g., a model based on Tensor operations) is another line of future work, alongside developing built-in support for ONNX networks. This will allow us to use our framework to formally show the semantic equivalence of these models. This will allow us to develop verified transformations that can be used to optimise neural networks for, e.g., making them easier to formally analyse or for improving their runtime performance.

Availability. The formalisation and case studies are available to view on Zenodo [9]. The materials inculde both the Isabelle/HOL implementation and the detailed documentation generated by Isabelle.

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., S. Corrado, G., 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., Mané, 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., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X.: *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. `https://www.tensorflow.org/`.
2. Abdulaziz, M., and Kurz, F.: Verified SAT-Based AI Planning. Archive of Formal Proofs (2020)

3. Aggarwal, C.C.: 'Machine learning with shallow neural networks'. In: Neural Networks and Deep Learning. Vol. 10. Springer, 2018. ISBN: 9783030068561.
4. Banerjee, K., C., V.P., Gupta, R.R., Vyas, K., H., A., and Mishra, B.: Exploring Alternatives to Softmax Function. (2020)
5. Barocas, S., Hardt, M., and Narayanan, A.: Fairness in machine learning. Nips tutorial 1, 2 (2017)
6. Bentkamp, A.: Expressiveness of Deep Learning. Archive of Formal Proofs (2016)
7. Bonaert, G., Dimitrov, D.I., Baader, M., and Vechev, M.: Fast and Precise Certification of Transformers. In: PLDI, pp. 466–481. ACM, Virtual, Canada (2021). DOI: 10.1145/3453483.3454056
8. Brucker, A.D.: Nano JSON: Working with JSON formatted data in Isabelle/HOL and Isabelle/ML. Archive of Formal Proofs (2022)
9. Brucker, A.D., and Stell, A.: *Dataset: Feedforward Neural Network Verification in Isabelle/HOL*. Dec. 2022. DOI: 10.5281/zenodo.7418170.
10. BS EN 50128:2011: Railway applications – Communication, signalling and processing systems – Software for railway control and protecting systems. Standard, British Standards Institute (BSI) (2014)
11. Campbell, A., Both, A., and Sun, Q.: Detecting and mapping traffic signs from Google Street View images using deep learning and GIS. Computers, Environment and Urban Systems 77, 101350 (2019). DOI: 10.1016/j.compenvurbsys.2019.101350
12. Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic 5(2), 56–68 (1940)
13. Cohen, N., Sharir, O., and Shashua, A.: On the expressive power of deep learning: A tensor analysis. In: Conference on learning theory, pp. 698–728 (2016)
14. *Common Criteria for Information Technology Security Evaluation (Version 3.1, Release 5)*. Available at https://www.commoncriteriaportal.org/cc/. 2017.
15. Dvijotham, K., Stanforth, R., Gowal, S., Mann, T.A., and Kohli, P.: A Dual Approach to Scalable Verification of Deep Networks. In: UAI, p. 3 (2018)
16. *ECMA-404:The JSON data interchange syntax*. https://www.ecma-international.org/publications-and-standards/standards/ecma-404/. Dec. 2017.
17. Ehlers, R.: Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In: D'Souza, D., and Narayan Kumar, K. (eds.) Automated Technology for Verification and Analysis, pp. 269–286. Springer, Cham (2017)
18. Goodfellow, I., Lee, H., Le, Q., Saxe, A., and Ng, A.: Measuring invariances in deep networks. Advances in neural information processing systems 22 (2009)
19. Harris, C.R., Millman, K.J., Walt, S.J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M.H. van, Brett, M., Haldane, A., Río, J.F. del, Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T.E.: Array programming with NumPy. Nature 585(7825), 357–362 (2020). DOI: 10.1038/s41586-020-2649-2
20. Huang, X., Kwiatkowska, M., Wang, S., and Wu, M.: Safety verification of deep neural networks. In: CAV, pp. 3–29 (2017)
21. Katz, G., Barrett, C.W., Dill, D.L., Julian, K., and Kochenderfer, M.J.: Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In: Majumdar, R., and Kuncak, V. (eds.) CAV. LNCS 10426, pp. 97–117. Springer (2017). DOI: 10.1007/978-3-319-63387-9_5
22. Katz, G., Huang, D.A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zelji, A., Dill, D.L., Kochenderfer, M.J., and Barret, C.: The

Marabou Framework for Verification and Analysis of Deep Neural Networks. In: CAV, pp. 443–452 (2019)
23. Klein, G.: Operating System Verification — An Overview. Sdhan 34(1), 27–69 (2009)
24. Krizhevsky, A., Sutskever, I., and Hinton, G.E.: ImageNet Classification with Deep Convolutional Neural Networks. Commun. ACM 60(6), 84–90 (2017). DOI: 10.1145/3065386
25. Kurd, Z., Kelly, T., and Austin, J.: Developing artificial neural networks for safety critical systems. Neural Computing and Applications 16(1), 11–19 (2007)
26. Matichuk, D., Murray, T., and Wenzel, M.: Eisbach: A Proof Method Language for Isabelle. Journal of Automated Reasoning 56(3), 261–282 (2016). DOI: 10.1007/s10817-015-9360-2
27. Mirman, M., Gehr, T., and Vechev, M.: Differentiable abstract interpretation for provably robust neural networks. In: Machine Learning, pp. 3578–3586 (2018)
28. Nipkow, T., Paulson, L.C., and Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic. Springer-Verlag, Heidelberg (2002). DOI: 10.1007/3-540-45949-9
29. Paulson, L.C.: ML for the Working Programmer. Cambridge Press (1996)
30. Pulina, L., and Tacchella, A.: An abstraction-refinement approach to verification of artificial neural networks. In: CAV, pp. 243–257 (2010)
31. Rintanen, J.: Madagascar: Scalable planning with SAT. IPC 21, 1–5 (2014)
32. Rumelhart, D.E., Hinton, G.E., and Williams, R.J.: Learning representations by back-propagating errors. nature 323(6088), 533–536 (1986)
33. Seshia, S.A., Desai, A., Dreossi, T., Fremont, D.J., Ghosh, S., Kim, E., Shivakumar, S., Vazquez-Chanlatte, M., and Yue, X.: Formal specification for deep neural networks. In: Automated Technology for Verification and Analysis, pp. 20–34 (2018)
34. Singh, G., Gehr, T., Püschel, M., and Vechev, M.: Boosting robustness certification of neural networks. In: Learning Representations (2018)
35. Smilkov, D., Thorat, N., Assogba, Y., Yuan, A., Kreeger, N., Yu, P., Zhang, K., Cai, S., Nielsen, E., Soergel, D., Bileschi, S., Terry, M., Nicholson, C., Gupta, S.N., Sirajuddin, S., Sculley, D., Monga, R., Corrado, G., Viégas, F.B., and Wattenberg, M.: TensorFlow.js: Machine Learning for the Web and Beyond. CoRR abs/1901.05350 (2019)
36. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R.: Intriguing properties of neural networks. In: International Conference on Learning Representations (2014)
37. Taigman, Y., Yang, M., Ranzato, M., and Wolf, L.: DeepFace: Closing the Gap to Human-Level Performance in Face Verification. In: 2014 IEEE Conference on Computer Vision and Pattern Recognition, pp. 1701–1708 (2014). DOI: 10.1109/CVPR.2014.220
38. Weng, L., Zhang, H., Chen, H., Song, Z., Hsieh, C.-J., Daniel, L., Boning, D., and Dhillon, I.: Towards fast computation of certified robustness for relu networks. In: Machine Learning, pp. 5276–5285 (2018)
39. Wenzel, M., and Wolff, B.: Building Formal Method Tools in the Isabelle/Isar Framework. In: TPHOLs 2007. Ed. by K. Schneider and J. Brandt, pp. 352–367. Springer-Verlag, Heidelberg (2007). DOI: 10.1007/978-3-540-74591-4_26
40. Wenzel, M., and Paulson, L.C.: Isabelle/Isar. In: Wiedijk, F. (ed.) The Seventeen Provers of the World, Foreword by Dana S. Scott. LNCS, vol. 3600, pp. 41–49. Springer, Heidelberg (2006). DOI: 10.1007/11542384_8