Formalizing Piecewise Affine Activation Functions of Neural Networks in Coq

Andrei Aleksandrov[0000−0002−4717−4206] and Kim Völlinger[0000−0002−8988−0053]

Technische Universität Berlin, Germany
andrei.aleksandrov@campus.tu-berlin.de
voellinger@tu-berlin.de

Abstract. Verification of neural networks relies on activation functions being *piecewise affine* (pwa) — enabling an encoding of the verification problem for theorem provers. In this paper, we present the first formalization of pwa activation functions for an interactive theorem prover tailored to verifying neural networks within Coq using the library Coquelicot for real analysis. As a proof-of-concept, we construct the popular pwa activation function ReLU. We integrate our formalization into a Coq model of neural networks, and devise a verified transformation from a neural network $\mathcal{N}$ to a pwa function representing $\mathcal{N}$ by composing pwa functions that we construct for each layer. This representation enables encodings for proof automation, e.g. Coq's tactic `lra` – a decision procedure for linear real arithmetic. Further, our formalization paves the way for integrating Coq in frameworks of neural network verification as a fallback prover when automated proving fails.

Keywords: Piecewise Affine Function · Neural Network · Interactive Theorem Prover · Coq· Verification.

1 Introduction

The growing importance of neural networks motivates the search of verification techniques for them. Verification with *automatic* theorem provers is vastly under study, usually targeting feedforward networks with *piecewise affine* (pwa) activation functions since the verification problem can be then encoded as an SMT or MILP problem. In contrast, few attempts exist on investigating *interactive* provers. Setting them up for this task though offers not only a fallback option when automated proving fails but also insight on the verification process.

That is why in this paper, we work towards this goal by presenting the first formalization of pwa activation functions for an interactive theorem prover tailored to verifying neural networks with Coq. We constructively define pwa functions using the polyhedral subdivision of a pwa function [25] since many algorithms working on polyhedra are known [26] with some tailored to reasoning about reachability properties of neural networks [30]. Motivated by verification, we restrict pwa functions by a polyhedron's constraint to be *non-strict* in order to suit linear programming [29] and by employing *finitely* many polyhedra to fit

SMT/MILP solvers [11,29]. We use reals supported by the library COQUELICOT to enable reasoning about gradients and matrices with COQ's standard library providing the tactic `lra` – a decision procedure for linear real arithmetic. As a proof-of-concept, we construct the activation function RELU– one of the most popular in industry [20] and formal verification [8]. Furthermore, we devise a sequential COQ model of feedforward neural networks integrating PWA activation layers. Most importantly, we present a verified transformation from a neural network $\mathcal{N}$ to a PWA function $f_\mathcal{N}$ representing $\mathcal{N}$ with the main benefit being again encodings for proof automation. To this end, we introduce two verified binary operations on PWA functions – usual function composition and an operator to construct a PWA function for each layer. In particular, we provide the following contributions with the corresponding COQ code available on GITHUB[1]:

1. a formalization of PWA functions based on polyhedral subdivision tailored to verification of neural networks (Section 3),
2. a construction of the popular activation function RELU (Section 3),
3. a sequential model for feedforward neural networks with parameterized layers (Section 4),
4. composition for PWA functions and an operator for constructing higher dimensional PWA functions out of lower dimensional ones (Section 4), and
5. a verified transformation from a feedforward neural network with PWA activation to a single PWA function representing the network (Section 4).

Related Work. A variety of work on using automatic theorem provers to verify neural networks exists with the vast majority targeting feedforward neural networks with PWA activation functions [6, 8, 12, 15, 18, 19, 24]. In comparison, little has been done regarding interactive theorem provers with some mechanized results from machine learning [2, 22], a result on verified training in LEAN [27] and, relevant to this paper, pioneering work on verifying networks in ISABELLE [7] and in COQ [3]. Apart from [7] targeting ISABELLE instead of COQ, both network models are not generalized by entailing a formalization of PWA functions and in addition they do not offer a model of the network as a (PWA) function – both contributions of this paper.

2   Preliminaries

We clarify notations and definitions important to this paper. We write $dom(f)$ for a function's domain, $dim(f)$ for the dimension of $dom(f)$ and $(f \circ g)(x)$ for function composition. For a matrix M, M^T is the transposed matrix. We consider block matrices. To clarify notation, consider a block matrix made out of matrices $M_1, ..., M_4$:

$$\left[\begin{array}{c|c} M_1 & M_2 \\ \hline M_3 & M_4 \end{array}\right]$$

---

[1] At `https://github.com/verinncoq/formalizing-pwa` with matrix_extensions.v (Section 2), piecewise_affine.v (Section 3.1), neuron_functions.v (Section 3.2), neural_networks.v (Section 4.1 and 4.4) and pwaf_operations.v (Section 4.2 and 4.3).

2.1 Piecewise Affine Topology

We give the important definitions regarding PWA functions [23, 25, 32].

Definition 1 (Linear Constraint). *For some $c \in \mathbb{R}^n, b \in \mathbb{R}$, a* linear constraint *is an inequality of form $c^T x \leq b$ for any $x \in \mathbb{R}^n$.*

Definition 2 (Polyhedron[2]). *A* polyhedron P *is the intersection of finitely many halfspaces, meaning $P := \{x \in \mathbb{R}^n | c_1^T x \leq b_1 \wedge \ldots \wedge c_m^T x \leq b_m\}$ with $c_i, b_i \in \mathbb{R}^n, b_i \in \mathbb{R}$ and $i \in \{1, \ldots, m\}$.*

We denote the constraints of P as $\mathcal{C}(P) := \{(c_1^T x \leq b_1), \ldots, (c_m^T x \leq b_m)\}$ for readability even though a constraint is given by c_i and b_i while x is arbitrary.

Definition 3 (Affine Function[3]). *A function $f : \mathbb{R}^m \to \mathbb{R}^n$ is called* affine *if there exists $M \in \mathbb{R}^{n \times m}$ and $b \in \mathbb{R}^n$ such that for all $x \in \mathbb{R}^m$ holds: $f(x) = Mx + b$.*

Definition 4 (Polyhedral Subdivision). *A* polyhedral subdivision $S \subseteq \mathbb{R}^n$ *is a finite set of polyhedra $\mathbf{P} := \{P_1, \ldots, P_m\}$ such that (1) $S = \bigcup_{i=1}^m P_i$ and (2) for all $P_i, P_j \in \mathbf{P}, x \in P_i \cap P_j$, and for all $\epsilon > 0$ there exists x' such that $|x - x'| < \epsilon$, and $x' \notin P_i \cap P_j$.*

Definition 5 (Piecewise Affine Function). *A continuous function $f : D \subseteq \mathbb{R}^m \to \mathbb{R}^n$ is* piecewise-affine *if there is a polyhedral subdivision $\mathbf{P} = \{P_1, \ldots, P_l\}$ of D and a set of affine functions $\{f_1, \ldots, f_l\}$ such that for all $x \in D$ holds $f(x) = f_i(x)$ if $x \in P_i$.*

2.2 Neural Networks

Neural networks approximate functions by learning from sample points during training [9] with arbitrary precision [10, 14, 16]. A feedforward neural network is a directed acyclic graph with the edges having weights and the vertices (neurons) having biases and being structured in layers. Each layer applies a generic affine function for summation and an activation function (possibly a PWA function). In many machine learning frameworks (e.g. PYTORCH), these functions are modelled as separate layers followed up by each other. We adopt this structure in our CoQ model with a *linear layer* implementing the generic affine function. Every network has an input and an output layer with optional hidden layers in between.

---

[2] In the literature often referred to as a convex, closed polyhedron.

[3] A linear function is a special case of an affine function [31]. However, in literature, the term linear is sometimes used for both.

2.3   Interactive Theorem Prover Coq & Library Coquelicot

We use the interactive theorem prover Coq [28] providing a non-turing-complete functional programming language extractable to selected functional programming languages and a proof development system – a popular choice for formal verification of programs and formalization of mathematical foundations. Additionally, we use the real analysis library Coquelicot [5] offering derivatives, integrals, and matrices compatible with Coq's standard library.

Extensions in Coq*: Column Vectors & Block Matrices.* For this paper, we formalized column vectors and block matrices on top of Coquelicot. A column vector `colvec` is identified with matrices and equipped with a dot product `dot` on vectors and some additional lemmas to simplify proofs. Additionally, we formalized several notions for Coquelicot's matrix type. We provide multiplication of a matrix with a scalar `scalar_mult` and transposition `transpose` of matrices. We provide operations on different shapes of matrices and vectors such as a right-to-left construction of block diagonal matrices `block_diag_matrix`, a specialization thereof on vectors `colvec_concat` and extensions of vectors with zeroes on the bottom `extend_colvec_at_bottom` or top `extend_colvec_on_top`, denoted as follows: $\left[\begin{array}{c|c} M_1 & 0 \\ \hline 0 & M_2 \end{array}\right]$, $\left[\begin{array}{c} \vec{v}_1 \\ \hline \vec{v}_2 \end{array}\right]$, $\left[\begin{array}{c} \vec{v} \\ \hline \vec{0} \end{array}\right]$, and $\left[\begin{array}{c} \vec{0} \\ \hline \vec{v} \end{array}\right]$. Moreover, we proved lemmas relating all new operations with each other and the existing matrix operations.

3   Formalization of Piecewise Affine Functions in Coq

We formalize PWA functions tailored to neural network verification with PWA activation. As a proof-of-concept, we construct the activation function Rectified Linear Unit (ReLU) – one of the most popular activation functions in industry [20] and formal verification [8].

3.1   Inductive Definition of PWA Functions

We define a linear constraint with a dimension *dim* and parameters, vector $c \in \mathbb{R}^{dim}$ and scalar $b \in \mathbb{R}$, being satisfied for a vector $x \in \mathbb{R}^{dim}$ if $c \cdot x \leq b$:

```
Inductive LinearConstraint (dim:nat) : Type :=
| Constraint (c: colvec dim)(b: R).

Definition satisfies_lc {dim: nat} (x: colvec dim) (l: LinearConstraint dim)
: Prop := match l with | Constraint c b ⇒ dot c x <= b end.
```

We define a polehydron as a finite set of linear constraints together with a predicate stating that a point lies in a polyhedron:

```
Inductive ConvexPolyhedron (dim: nat) : Type :=
| Polyhedron (constraints: list (LinearConstraint dim)).
```

```
Definition in_convex_polyhedron {dim: nat} (x: colvec dim) (p:
    ConvexPolyhedron dim) :=
match p with | Polyhedron lcs ⇒
  forall constraint, In constraint lcs → satisfies_lc x constraint end.
```

Finally, we define a PWA function as a record composed of the fields `body` holding the polyhedral subdivision for piecewise construction of the function, and `prop` for the property univalence (i.e. all "pieces" together yield a function).

```
Record PWAF (in_dim out_dim: nat): Type := mkPLF {
    body: list (ConvexPolyhedron in_dim * ((matrix out_dim in_dim) * colvec
        out_dim));
    prop: pwaf_univalence body; }.
```

Piecewise Construction. We construct a PWA function f by a list of polyhedra, matrices and vectors with a triple (P, M, b) defining a "piece" of f by an affine function with $f(x) = Mx + b$ if $x \in P$. For evaluation, we search a polyhedron containing x and compute the affine function:

```
Fixpoint pwaf_eval_helper {in_dim out_dim: nat}
    (body: list (ConvexPolyhedron in_dim * ((matrix (T:=R) out_dim in_dim) *
        colvec out_dim))) (x: colvec in_dim)
    : option (ConvexPolyhedron in_dim * ((matrix out_dim in_dim) * colvec
        out_dim)) :=
    match body with
    | nil ⇒ None
    | body_el :: next ⇒
        match body_el with
        | (polyh, (M, b)) ⇒
            match polyhedron_eval x polyh with
            | true ⇒ Some (body_el)
            | false ⇒ pwaf_eval_helper next x
    end end end.
```

To handle the edge case where no such polyhedron is found (i.e. $x \notin dom(f)$), we use a wrapper function `pwaf_eval`. For the purpose of proving, we define a predicate `in_pwaf_domain` for the existence of such a polyhedron and a predicate `is_pwaf_value` for a stating the function is evaluated to a certain value.

Univalence. We enforce the construction to be a function by stating univalence, in this case all pairs of polyhedra having coinciding affine functions in their intersection, requiring a proof for each instance of type `PWAF`:

```
Definition pwaf_univalence {in_dim out_dim: nat}
    (l: list (ConvexPolyhedron in_dim *
      ((matrix out_dim in_dim) * colvec out_dim))) :=
    ForallPairs (fun e1 e2 ⇒ let p1 := fst e1 in let p2 := fst e2 in
```

```
forall x, in_convex_polyhedron x p1 ∧ in_convex_polyhedron x p2 →
  let M1 := fst (snd e1) in let b1 := snd (snd e1) in
  let M2 := fst (snd e2) in let b2 := snd (snd e2) in
    Mplus (Mmult M1 x) b1 = Mplus (Mmult M2 x) b2 ) l.
```

Class of Formalized PWA Functions. Motivated by PWA activation functions in the context of neural network verification, our PWA functions are restricted by

(1) all linear constraints being *non-strict*, and
(2) being defined over a union of *finitely* many polyhedra.

Restriction (1) is motivated by linear programming usually dealing with non-strict constraints [29], and restriction (2) by MILP/SMT solvers commonly accepting finitely many variables [11, 29]. Since we use that every continuous PWA function on $\mathbb{R}^n$ admits a polyhedral subdivision of the domain [25], all continuous PWA functions with a finite subdivision can be encoded.

   For PWA functions not belonging to this class, consider any discontinuous PWA function since discontinuity violates restriction (1), and any periodic PWA function as excluded by restriction (2) due to having infinitely many "pieces".

Choice of Formalization. We use real numbers (instead of e.g. rationals or floats) to enable COQUELICOT's reasoning about derivatives interesting for neural networks' gradients. COQUELICOT builds up on the reals of COQ's standard library allowing the use of COQ's tactic `lra` – a COQ-native decision procedure for linear real arithmetic.

   Moreover, we use inductive types since they come with an induction principle and therefore ease proving. In addition, the type `list` (e.g. used for the definition of PWA functions) enjoys extensive support in COQ. For example, `pwaf_univalence` is stated using the list predicate `ForAllPairs` and proofs intensively involve lemmas from COQ's standard library.

   A constructive definition using the polyhedral subdivision is interesting since many efficient algorithms are known that work on polyhedra [26] with even algorithms tailored to neural network verification around [30]. We expect that such algorithms are implementable in an idiomatic functional style using our model. Furthermore, we anticipate easy-to-implement encodings for proof automation.

3.2   Example: Rectified Linear Unit Activation Function

We construct RELU as a PWA function defined by two "pieces" each of which being a linear function. The function is defined as:

$$\mathrm{ReLU}(x) := \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$$

Piecewise Construction. The intervals, $(-\infty, 0)$ and $[0, \infty)$, each correspond to a polyhedron in $\mathbb{R}$ defined by a single constraint[4]: $P_{left} := \{x \in \mathbb{R}^1 | [1] \cdot x <= 0\}$ and $P_{right} := \{x \in \mathbb{R}^1 | [-1] \cdot x <= 0\}$. We define these polyhedra as follows:[5]

```
Definition ReLU1d_polyhedra_left := Polyhedron 1 [Constraint 1 Mone 0].
Definition ReLU1d_polyhedra_right
    := Polyhedron 1 [Constraint 1 (scalar_mult (−1) Mone) 0].
```

ReLU's construction list contains these polyhedra each associated with a matrix and vector, in these cases $([0], [0])$ and $([1], [0])$, for the affine functions:

```
Definition ReLU1d_body: list (ConvexPolyhedron 1 * (matrix (T:=R) 1 1 *
    colvec 1))
  := [(ReLU1d_polyhedra_left, (Mzero, null_vector 1));
      (ReLU1d_polyhedra_right, (Mone, null_vector 1))].
```

Univalence. Note that while ReLU's intervals are distinct, the according polyhedra with non-strict constraints are not. To ensure the construction to be a function, we prove univalence by proving that only $[0] \in (P_{left} \cap P_{right})$:

```
Lemma RelU1d_polyhedra_intersect_0:
    forall x, in_convex_polyhedron x ReLU1d_polyhedra_left ∧
        in_convex_polyhedron x ReLU1d_polyhedra_right → x = null_vector 1.
```

Finally, we ensure for each polyhedra pair holds $[1] \cdot [0] + [0] = [0] \cdot [0] + [0]$, and instantiate a `PWAF` by `Definition ReLU1dPWAF := mkPLF 1 1 ReLU1d_body ReLU1d_pwaf_univalence`.

On the Construction of pwa *Functions.* Analogously to the ReLU example, other activation functions sharing its features of being one-dimensional and consisting of a few polyhedra can be constructed similarly. We can construct a multi-dimensional version out of a one-dimensional function as we will illustrate for ReLU in Section 4.3. Activation functions that require more effort to construct are for example different types of pooling [9], mostly due to a non-trivial polyhedra structure and inherent multi-dimensionality. This effort motivates future development of more support in constructing pwa functions with the goal to compile a library of layer types.

---

[4] Matrices involved are one-dimensional vectors since ReLU is one-dimensional. For technical reasons, in Coq, the spaces $\mathbb{R}$ and $\mathbb{R}^1$ differ with the latter working on one-dimensional vectors instead on scalars.

[5] Mone is Coquelicot's identity matrix which in this case is a one-dimensional vector. scalar_mult is multiplication of a matrix by a scalar (see Section 2).

4   Verified Transformation of a Neural Network to a PWA Function

We present our main contribution: a formally verified transformation of a feed-forward neural network with PWA activations into a single PWA function. First, we introduce a COQ model for feedforward neural networks (Section 4.1). We follow up with two verified binary operations on PWA functions at the heart of the transformation, *composition* (Section 4.2) and *concatenation* (Section 4.3), and finish with the verified transformation (Section 4.4).

4.1   Neural Network Model in COQ

We define a neural network *NNSequential* as a list-like structure containing layers parameterized on the type of activation, and the input's, output's and hidden layer's dimensions with dependent types preventing dimension mismatch:

```
Inductive NNSequential {input_dim output_dim: nat} :=
| NNOutput : NNSequential
| NNPlainLayer {hidden_dim: nat}:
    (colvec input_dim → colvec hidden_dim)
    → NNSequential (input_dim:=hidden_dim) (output_dim:=output_dim)
    → NNSequential
| NNPWALayer {hidden_dim: nat}:
    PWAF input_dim hidden_dim
    → NNSequential (input_dim:=hidden_dim) (output_dim:=output_dim)
    → NNSequential
| NNUnknownLayer {hidden_dim: nat}:
    NNSequential (input_dim:=hidden_dim) (output_dim:=output_dim)
    → NNSequential.
```

The network model has four layer types: `NNOutput` as the last layer propagates input values to the output; `NNPlainLayer` is a layer allowing any function in COQ defined on real vectors; `NNPWALayer` is a PWA activation layer – the primary target of our transformation; and `NNUnknownLayer` is a stub for a layer with an unknown function.

Informally speaking, the semantics of our model is as follows: for a layer `NNOutput` the identity function[6] is evaluated, for `NNPlainLayer` the passed function, for `NNPWALayer` the passed PWA function, and for `NNUnknownLayer` a failure is raised. Thus, the *NNSequential* type does not prescribe any specific functions of layers but expects them as parameters.

An Example of a Neural Network. In order to give an example, we define specific layers for a network, in this case the PWA layers LINEAR and RELU:

```
Definition NNLinear {input_dim hidden_dim output_dim: nat}
```

---

[6] We use the customized identity function *flex_dim_copy*.

```
  (M: matrix hidden_dim input_dim) (b: colvec hidden_dim)
  (NNnext: NNSequential (input_dim:=hidden_dim) (output_dim:=output_dim))
  := NNPWALayer (LinearPWAF M b) NNnext.

Definition NNReLU {input_dim output_dim: nat}
  (NNnext: NNSequential (input_dim:=input_dim) (output_dim:=output_dim))
  := NNPWALayer (input_dim:=input_dim) ReLU_PWAF NNnext.
```

As an example, we consider a neural network with these two layers:

```
Definition example_weights: matrix 2 2 := [[2.7, 0],[1, 0.01]].
Definition example_biases: colvec 2 := [[1], [0.25]].
Definition example_nn := (NNLinear example_weights example_biases
                         (NNReLU (NNOutput (output_dim:=2)))).
```

From a Trained Neural Network into the World of Coq. As illustrated, we can construct feedforward neural networks in Coq. Another option is to convert a neuronal network trained outside of Coq into an instance of the model. In [3] a python script is used for conversion from PyTorch to their Coq model without any correctness guarantess, while in [7] an import mechanism from TensorFlow into Isabelle is used, where correctness of the import has to be established for each instance of their model. We are working with a converter expecting a neural network in the ONNX format (i.e. a format for neural network exchange supported by most frameworks) [4] to produce an according instance in our Coq model [13].[7] This converter is mostly written within Coq with its core functionality being verified.

Choice of Model. While feedforward neural networks are often modeled as directed acyclic graphs [1, 17], in the widely used machine learning frameworks TensorFlow and PyTorch a sequential model of layers is employed as well. Our model corresponds to the latter, and is inspired by the, to our knowledge, only published neural network model in Coq (having been used for generalization proofs) [3]. Our model though is more generic by having parameterized layers instead of being restricted to ReLU activation. Moreover, while their model works with customized floats, we decided for reals in order to support Coquelicot's real analysis as discussed in Section 3.

A graph-based model carries the potential to be extended to other types of neural networks such as recurrent networks featuring loops in the length of the input. For the reason of being generic, ONNX employs a graph-based model. Hence, an even more generic graph-based Coq model is in principle desirable but it also adds complexity. In [7] the focus is on a sequential model which the authors showed to be superior to a graph-based model for the purpose of verification. Hence, we expect that the need for a sequential Coq model for *feedforward* networks will stay even in the existence of a graph-based model.

---

[7] A bachlor thesis supervised by one of the authors and scheduled for publication.

4.2   Composition of PWA functions

Besides composition being a general purpose binary operation closed over PWA functions [25], it is needed in our transformation to compose PWA layers. Since for PWA functions $f : \mathbb{R}^l \to \mathbb{R}^n$ and $g : \mathbb{R}^m \to \mathbb{R}^l$ their composition $z = f \circ g$ is a PWA function, composition in Coq produces an instance of type PWAF requiring a construction and a proof of univalence:

```
Definition pwaf_compose {in_dim hidden_dim out_dim: nat}
    (f: PWAF hidden_dim out_dim) (g: PWAF in_dim hidden_dim)
    : PWAF in_dim out_dim := mkPLF in_dim out_dim
        (pwaf_compose_body f g) (pwaf_compose_univalence f g).
```

Piecewise Construction of Composition. Assume a PWA function f defined on the polyhedra set $\mathbf{P}^f = \{P_1^f, \ldots, P_k^f\}$ with affine functions given by the parameter set $\mathbf{A}^f = \{(M_1^f, b_1^f), \ldots, (M_K^f, b_k^f)\}$. Analogously, g is given by $\mathbf{P}^g$ and $\mathbf{A}^g$. For computing a composed function $z = f \circ g$ at any $x \in \mathbb{R}^m$, we need a polyhedron $P_j^g \in \mathbf{P}^g$ such that $x \in P_j^g$ to compute $g(x) = M_j^g x + b_j^g$ with $(M_j^g, b_j^g) \in \mathbf{A}^g$. Following, we need a polyhedron $P_i^f \in \mathbf{P}^f$ with $g(x) \in P_i^f$ to finally compute $z(x) = M_i^f g(x) + b_i^f$ with $(M_i^f, b_i^f) \in \mathbf{A}^f$.

We have to reckon on function composition on the level of polyhedra sets to construct z's polyhedra set $\mathbf{P}^z$. For each pair $P_i^f \in \mathbf{P}^f$, $P_j^g \in \mathbf{P}^g$, we create a polyhedron $P_{i,j}^z \in \mathbf{P}^z$ such that $x \in P_{i,j}^z$ iff $x \in P_j^g$ and $M_j^g x + b_j^g \in P_i^f$ with $(M_j^g, b_j^g) \in \mathbf{A}^g$. Consequently, $\mathcal{C}(P_j^g) \subseteq \mathcal{C}(P_{i,j}^z)$ while the constraints of P_i^f have to be modified. For $(c_i \cdot x \le b_i) \in \mathcal{C}(P_i^f)$ we have the modified constraint $((c_i^T M_j^g) \cdot x \le b_i - c_i \cdot b_j^g) \in \mathcal{C}(P_{i,j}^z)$. We construct a polyhedra set accordingly in Coq including empty polyhedra in case no qualifying pair of polyhedra exists:

```
Fixpoint compose_polyhedra_helper {in_dim hidden_dim: nat}
    (M: matrix hidden_dim in_dim) (b1: colvec hidden_dim)
    (l_f: list (LinearConstraint hidden_dim)) :=
    match l_f with
    | [] ⇒ []
    | (Constraint c b2) :: tail ⇒
        Constraint in_dim
            (transpose (Mmult (transpose c) M)) (b2 − (dot c b1)) ::
                compose_polyhedra_helper M b1 tail
    end.

Definition compose_polyhedra {in_dim hidden_dim: nat}
    (p_g: ConvexPolyhedron in_dim)
    (M: matrix hidden_dim in_dim) (b: colvec hidden_dim)
    (p_f: ConvexPolyhedron hidden_dim) :=
    match p_g with | Polyhedron l1 ⇒
        match p_f with | Polyhedron l2 ⇒
            Polyhedron in_dim (l1 ++ compose_polyhedra_helper M b l2)
```

```
      end end.
```

Further, each $(M_{i,j}^z, b_{i,j}^z) \in \mathbf{A}^z$ is defined as $(M_j^f M_i^g, M_j^f b_i^g + b_j^f)$ as a result of usual composition of two affine functions:

```
Definition compose_affine_functions {in_dim hidden_dim out_dim: nat}
    (M_f: matrix (T:=R) out_dim hidden_dim) (b_f: colvec out_dim)
    (M_g: matrix (T:=R) hidden_dim in_dim) (b_g: colvec hidden_dim) :=
    (Mmult M_f M_g, Mplus (Mmult M_f b_g) b_f).
```

Univalence of Composition. Due to the level of details, the Coq proof for the composed function z satisfying univalence is not included in this paper (see Theorem `pwaf_compat_univalence`).

Composition Correctness. For establishing the correctness of the composition, we proved the following theorem:

```
Theorem pwaf_compose_correct:
    forall in_dim hid_dim out_dim x f_x g_x
        (f: PWAF hid_dim out_dim) (g: PWAF in_dim hid_dim),
        in_pwaf_domain g x → is_pwaf_value g x g_x →
        in_pwaf_domain f g_x → is_pwaf_value f g_x f_x →
        let fg := pwaf_compose f g in
        in_pwaf_domain fg x ∧ is_pwaf_value fg x f_x.
```

For one of the lemmas (`compose_polyhedra_subset_g`) we proved that polyhedra of g are only getting smaller by composing g with f while the borders that are set by polyhedra of g being kept.

4.3   Concatenation: Layers of Neural Networks as PWA Functions

While some neural networks come with each layer being *one* multi-dimensional function, many neural networks feature layers where each neuron is assigned the same lower dimensional function independently then applied to each neuron's input. Motivated by the transformation of a neural network into a single PWA function, we introduce a binary operation *concatenation* that constructs a single PWA function for each PWA layer of a neural network. Otherwise, concatenation is interesting due to constructing a multi-dimensional PWA function being challenging since a user has to define multiple polyhedra with a significant number of constraints. For illustration, we construct a multi-dimensional ReLU layer.

Concatenation of PWA functions has to yield an instance of type `PWAF` since being closed over PWA functions. Concatenation is defined as follows:

Definition 6 (Concatenation). *Let* $f : \mathbb{R}^m \to \mathbb{R}^n$ *and* $g : \mathbb{R}^k \to \mathbb{R}^l$. *The concatenation* $\oplus$ *is defined as:*

$$(f \oplus g)\left(\begin{bmatrix} x^f \\ x^g \end{bmatrix}\right) := \begin{bmatrix} f(x^f) \\ g(x^g) \end{bmatrix}$$

Piecewise Construction of Concatenation. Assume some $f, g, \mathbf{P}^f, \mathbf{P}^g, \mathbf{A}^f$ and $\mathbf{A}^g$ as previously used, and $z = f \oplus g$. The polyhedra set $\mathbf{P}^z$ contains the pairwise joined polyhedra of $\mathbf{P}^f$ and $\mathbf{P}^g$ but with each constraint of a polyhedron lifted to the dimension of z's domain. Consider a pair $P_i^f \in \mathbf{P}^f$ and $P_j^g \in \mathbf{P}^g$. For constraints $(c_i^f \cdot x^f \leq b_i^f) \in \mathcal{C}(P_i^f)$ and $(c_j^g \cdot x^g \leq b_j^g) \in \mathcal{C}(P_j^g)$ with $\begin{bmatrix} x^f \\ x^g \end{bmatrix} \in \mathbb{R}^{dim(f)+dim(g)}$, the following higher dimensional constraints are in $\mathcal{C}(P_{i,j}^z)$ with $P_{i,j}^z \in \mathbf{P}^z$: $\begin{bmatrix} c_i^f \\ 0 \end{bmatrix} \cdot \begin{bmatrix} x^f \\ x^g \end{bmatrix} \leq b_i^f$ and $\begin{bmatrix} 0 \\ c_j^g \end{bmatrix} \cdot \begin{bmatrix} x^f \\ x^g \end{bmatrix} \leq b_j^g$. Thus, we get $\begin{bmatrix} x^f \\ x^g \end{bmatrix} \in P_{i,j}^z$ iff $x^f \in P_i^f$ and $x^g \in P_j^g$.

Hence, the concatenation requires the pairwise join of all polyhedra $\mathbf{P}^f$ and $\mathbf{P}^g$ each with their constraints lifted to the higher dimension of z's domain:

```
Definition concat_polyhedra {in_dim1 in_dim2: nat}
    (p_f: ConvexPolyhedron in_dim1) (p_g: ConvexPolyhedron in_dim2):
    ConvexPolyhedron (in_dim1 + in_dim2) :=
    match p_f with | Polyhedron l1 ⇒
        match p_g with | Polyhedron l2 ⇒
            Polyhedron (in_dim1 + in_dim2)
                (extend_lincons_at_bottom l1 (in_dim1 + in_dim2) ++
                extend_lincons_on_top l2 (in_dim1 + in_dim2))
    end end.
```

The CoQ code uses two functions for insertion of zeros similar to the dimension operations (see Section 2). The corresponding affine function of $P_{i,j}^z$ is then:

$$(M_{i,j}^z, b_{i,j}^z) := (\begin{bmatrix} M_i^f & 0 \\ 0 & M_j^g \end{bmatrix}, \begin{bmatrix} b_i^f \\ b_j^g \end{bmatrix}).$$

Univalence of Concatenation. The technical proof of concatenation being univalent is outside of the scope of this paper (see `pwaf_concat_univalence`).

Concatenation Correctness. We proved the correctness of the concatenation:

```
Theorem pwaf_concat_correct:
    forall in_dim1 in_dim2 out_dim1 out_dim2 x1 x2 f_x1 g_x2
      (f: PWAF in_dim1 out_dim1) (g: PWAF in_dim2 out_dim2),
      in_pwaf_domain f x1 → is_pwaf_value f x1 f_x1 →
      in_pwaf_domain g x2 → is_pwaf_value g x2 g_x2 →
      let fg   := pwaf_concat f g in
      let x    := colvec_concat x1 x2 in
      let fg_x := colvec_concat f_x1 g_x2 in
      in_pwaf_domain fg x ∧ is_pwaf_value fg x fg_x.
```

The proof relies on an extensive number of lemmas connecting matrix operations to block matrices and vector reshaping.

Example: ReLU *Layer.* Using concatenation, we construct a multi-dimensional ReLU layer using one-dimensional ReLU (see Section 4.1). To construct a ReLU layer $\mathbb{R}^n \to \mathbb{R}^n$, we perform n concatenations of one-dimensional ReLU:

```
Fixpoint ReLU_PWAF_helper (in_dim: nat): PWAF in_dim in_dim :=
    match in_dim with
    | 0 ⇒ OutputPWAF (in_dim:=0) (out_dim:=0)
    | S n ⇒ pwaf_concat ReLU1dPWAF (ReLU_PWAF_helper n)
    end.
```

4.4   Transforming a Neural Network into a PWA Function

Building up on previous efforts, the transformation of a feedforward neural network with PWA activation functions into a single PWA function is straightforward. Using concatenation, we construct multi-dimensional PWA layers and then compose them to one PWA function representing the whole neural network. The transformation is illustrated conceptually in Figure 1.

The transformation in Coq simply fails when applied to hidden layers that are non-PWA:

```
Fixpoint transform_nn_to_pwaf {in_dim out_dim: nat}
    (nn: NNSequential (input_dim := in_dim) (output_dim := out_dim))
    : option (PWAF in_dim out_dim) :=
    match nn with
        | NNOutput ⇒ Some (OutputPWAF)
        | NNPlainLayer _ _ _ ⇒ None
        | NNUnknownLayer _ _ ⇒ None
        | NNPWALayer _ pwaf next ⇒
            match transform_nn_to_pwaf next with
            | Some next_pwaf ⇒ Some (pwaf_compose next_pwaf pwaf)
            | None ⇒ None
    end end.
```

Correctness of Transformation. For this transformation, we have also proven the following theorem in Coq to establish its correctness:

```
Theorem transform_nn_to_pwaf_correct:
    forall in_dim out_dim (x: colvec in_dim) (f_x: colvec out_dim) nn
        nn_pwaf,
        Some nn_pwaf = transform_nn_to_pwaf_correct nn →
        in_pwaf_domain nn_pwaf x →
        is_pwaf_value nn_pwaf x f_x ↔ nn_eval nn x = Some f_x.
```

For a neural network $\mathcal{N}$ and its transformed PWA function $f_{\mathcal{N}}$, the theorem states that for all inputs $x \in dom(f_{\mathcal{N}})$ holds $f_{\mathcal{N}}(x) = \mathcal{N}(x)$. The proof of this theorem relies on several relatively simple properties of the composition. Note that for
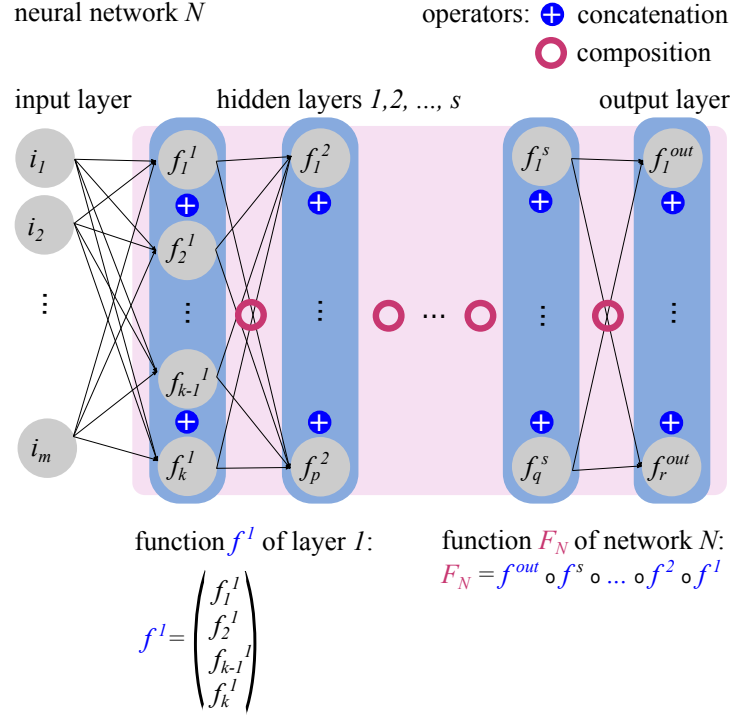
Fig. 1. Transformation of a feedforward network N with PWA activation functions into its representation as a PWA function F_N by concatenating neuron activation within each layer followed up by composing PWA layers.

$dom(f_\mathcal{N}) = \emptyset$ the theorem trivially holds, and in fact an additional proof is required for $f_\mathcal{N}$'s polyhedra being a subdivision of $dom(\mathcal{N})$ (i.e. $dom(f_\mathcal{N}(x)) = dom(\mathcal{N}(x)))$.

On the Representation of a Neural Network as a PWA *Function.* The main benefit of having a PWA function obtained from neural network lies in the option to use simple-to-implement encodings of PWA functions for different solvers, e.g. CoQ's tactic `lra` or MILP/SMT solvers. Hence, this representation paves the way for proof automation when stating theorems about the input-output relation of a network in CoQ.

Furthermore, a representation as a PWA function moves the structural complexity of a neural network to the polyhedral subdivision of the PWA function. This is interesting since local search can be applied to the set of polyhedra for reasoning about reachability properties in neural networks [30]. Furthermore, one may estimate the size of a PWA function's polyhedral subdivision for different architectures of neural network [21].

5   Discussion

We were working towards neural network verification in Coq with a verified transformation from a network to a PWA function being the main contribution.

Summary. We presented the first formalization of PWA activation functions for an interactive theorem prover tailored to verifying neural networks with Coq. For our constructive formalization, we used a PWA function's polyhedral subdivision due to the numerous efficient algorithms working on polyhedra. Our class of PWA functions is on-purpose restricted to suit linear programming by using non-strict constraints and to fit SMT/MILP solvers by employing finitely many polyhedra. Using Coquelicot's reals, we enabled reasoning about gradients and support Coq's tactic `lra`. With ReLU, we constructed one of the most popular activation functions. We presented a verified transformation from a neural network to its representation as a PWA function enabling encodings for proof automation for theorems about the input-output relation. To this end, we devised a sequential model of neural networks, and introduced two verified binary operation on PWA functions – usual function composition together with an operator to construct a PWA function for each layer.

Future Work. Since the main benefit of having a PWA function obtained from neural network lies in the many available encodings [8, 12] targeting different solvers, we envision encodings for our network model. These encodings have to be adapted to the verification within Coq with our starting point being the tactic `lra` – a Coq-native decision procedure for linear real arithmetic.

Moreover, moving the structural complexity of a neural network to the polyhedral subdivision of a PWA function, opens up on investigating algorithms working on polyhedra for proof automation with our main candidate being local search on polyhedra for reasoning about reachability properties in neural networks [30].

Further, for our model of neural networks, we intend a library of PWA activation functions with proof automation to ease construction. We also plan on a generic graph-based model for neural networks in Coq but as argued, we expect the sequential model to stay the mean of choice for feedforward networks. Additionally, since tensors are used in machine learning to incorporate complex mathematical operations, we aim to integrate a formalization of tensors tailored to neural network verification.

References

1. Aggarwal, C.C.: Neural Networks, pp. 211–251. Springer International Publishing, Cham (2021)
2. Alexander Bentkamp, J.B., Klakow, D.: A Formal Proof of the Expressiveness of Deep Learning. In: Journal of Automated Reasoning. Springer Verlag (2019). https://doi.org/10.1007/s10817-018-9481-5

3. Bagnall, A., Stewart, G.: Certifying the True Error: Machine Learning in Coq with Verified Generalization Guarantees. In: AAAI Conference on Artificial Intelligence (2019)
4. Bai, J., Lu, F., Zhang, K., et al.: ONNX: Open Neural Network Exchange. `https://github.com/onnx/onnx` (2019)
5. Boldo, S., Lelay, C., Melquiond, G.: Coquelicot: A User-Friendly Library of Real Analysis for Coq. Mathematics in Computer Science **9**(1), 41–62 (2015)
6. Botoeva, E., Kouvaros, P., Kronqvist, J., Lomuscio, A., Misener, R.: Efficient Verification of ReLU-Based Neural Networks via Dependency Analysis. Proceedings of the AAAI Conference on Artificial Intelligence **34**, 3291–3299 (04 2020). `https://doi.org/10.1609/aaai.v34i04.5729`
7. Brucker, A.D., Stell, A.: Verifying Feedforward Neural Networks for Classification in Isabelle/HOL. In: Proceedings of the 25th International Symposium on Formal Methods. Springer (2023), to appear
8. Bunel, R., Turkaslan, I., Torr, P.H., Kohli, P., Kumar, M.P.: A Unified View of Piecewise Linear Neural Network Verification. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems. p. 4795–4804. NIPS'18, Curran Associates Inc., Red Hook, NY, USA (2018)
9. Calin, O.: Deep Learning Architectures: A Mathematical Approach. Springer (2020)
10. Cybenko, G.: Approximation by superpositions of a sigmoidal function. Mathematics of Control, Signals and Systems **2**(4), 303–314 (1989)
11. De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Communications of the ACM **54**(9), 69–77 (2011)
12. Ehlers, R.: Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In: Automated Technology for Verification and Analysis (2017)
13. Gummersbach, L.: Ein verifizierter Converter für neuronale Netze von ONNX nach Coq. Bachelor's thesis, Technische Universität Berlin (2023), to appear at Technische Universität Berlin
14. Hanin, B.: Universal Function Approximation by Deep Neural Nets with Bounded Width and ReLU Activations. Mathematics **7**(10) (2019). `https://doi.org/10.3390/math7100992`
15. H.D. Tran, S. Bak, W.X., Johnson, T.: Verification of Deep Convolutional Neural Networks Using ImageStars. In: Lahiri, S., Wang, C. (eds.) Computer Aided Verification. Springer, Cham (2020). `https://doi.org/https://doi.org/10.1007/978-3-030-53288-8_2`
16. Hornik, K.: Approximation capabilities of multilayer feedforward networks. Neural networks **4**(2), 251–257 (1991)
17. Kruse, R., Mostaghim, S., Borgelt, C., Braune, C., Steinbrecher, M.: General Neural Networks, pp. 39–52. Springer International Publishing, Cham (2022)
18. Lin, W., Yang, Z., Chen, X., Zhao, Q., Li, X., Liu, Z., He, J.: Robustness Verification of Classification Deep Neural Networks via Linear Programming. pp. 11410–11419 (06 2019). `https://doi.org/10.1109/CVPR.2019.01168`
19. Liu, C., Arnon, T., Lazarus, C., Strong, C., Barrett, C., Kochenderfer, M.J.: Algorithms for Verifying Deep Neural Networks. Found. Trends Optim. **4**(3–4), 244–404 (feb 2021). `https://doi.org/10.1561/2400000035`
20. Montesinos López, Osval Antonio, A.M.L., Crossa, J.: Fundamentals of Artificial Neural Networks and Deep Learning. In: Multivariate Statistical Machine Learning Methods for Genomic Prediction. p. 379–425. Springer International Publishing (2022). `https://doi.org/https://doi.org/10.1007/978-3-030-89010-0_10`

21. Montúfar, G., Pascanu, R., Cho, K., Bengio, Y.: On the Number of Linear Regions of Deep Neural Networks. In: Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2. p. 2924–2932. NIPS'14, MIT Press, Cambridge, MA, USA (2014)
22. Murphy, C., Gray, P., Stewart, G.: Verified Perceptron Convergence Theorem. In: Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. p. 43–50. MAPL 2017, Association for Computing Machinery, New York, NY, USA (2017). `https://doi.org/10.1145/3088525.3088673`
23. Rourke, C., Sanderson, B.: Introduction to Piecewise-Linear Topology. Springer Berlin, Heidelberg (1982)
24. Scheibler, K., Winterer, L., Wimmer, R., Becker, B.: Towards Verification of Artificial Neural Networks. In: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (2015)
25. Scholtes, S.: Introduction to Piecewise Differentiable Equations. Springer New York (2012)
26. Schrijver, A.: Combinatorial Optimization: Polyhedra and Efficiency. Springer Berlin, Heidelberg (2002)
27. Selsam, D., Liang, P., Dill, D.L.: Developing bug-free machine learning systems with formal mathematics. In: Proceedings of the 34th International Conference on Machine Learning - Volume 70. p. 3047–3056. ICML'17, JMLR.org (2017)
28. Team, T.C.D.: The Coq Proof Assistant (Sep 2022). `https://doi.org/10.5281/zenodo.7313584`, `https://doi.org/10.5281/zenodo.7313584`
29. Vanderbei, R.J.: Linear Programming: Foundations and Extensions. Springer (2020)
30. Vincent, J.A., Schwager, M.: Reachable Polyhedral Marching (RPM): A Safety Verification Algorithm for Robotic Systems with Deep Neural Network Components. In: 2021 IEEE International Conference on Robotics and Automation (ICRA). pp. 9029–9035 (2021). `https://doi.org/10.1109/ICRA48506.2021.9561956`
31. Yang, X.S.: Mathematical foundations. In: Yang, X.S. (ed.) Introduction to Algorithms for Data Mining and Machine Learning, pp. 19–43. Academic Press (2019). `https://doi.org/https://doi.org/10.1016/B978-0-12-817216-2.00009-0`
32. Ziegler, G.M.: Lectures on Polytopes. Springer-Verlag New York, Inc. 1995 (1995)