

Formalizing Neural Networks

Achim D. Brucker[Ⓢ]

Amy Stell[Ⓢ]

April 10, 2026

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

Abstract

Deep learning, i.e., machine learning using neural networks, is used successfully in many application areas. Still, their use in safety-critical or security-critical applications is limited, due to the lack of testing and verification techniques.

We address this problem by formalizing an important class of neural networks, feed-forward neural networks, in Isabelle/HOL. We present two different approaches of formalizing feed-forward networks and show their equivalence as well as demonstrate their use in verifying certain safety and correctness properties of various example. Moreover, we do not only provide a formal model that allows to reason over feed-forward neural networks, we also provide a datatype package for Isabelle/HOL that supports importing models from TensorFlow.js.

Keywords: Deep Learning, Neural Networks, Verification, TensorFlow

Contents

1	Introduction	7
2	Preliminaries	13
2.1	Proofs and Definitions that Enrich the Matrix Formalization (Matrix_Utils)	13
2.1.1	List Properties	13
2.1.2	Vector and Matrix Properties	13
2.2	Infrastructure for Importing TensorFlow Models (TensorFlow_Import)	17
2.2.1	Encoder	18
2.2.2	Example Import	21
2.3	Common Infrastructure (NN_Common)	25
2.3.1	Utility Functions	25
2.3.2	Data Import	26
2.3.3	Common Infrastructure for Proof Tactics	26
3	Activation Functions	29
3.1	Defining Activation Functions and Their Derivatives (Activation_Functions)	29
3.1.1	Activation Functions	29
3.1.2	Derivatives of Activation Functions	33
3.1.3	Single Class Folding Activation Functions	33
3.1.4	Multiclass Folding Activation Functions	35
3.2	Encoding of Activation Functions (Activation_Functions)	36
4	Neural Networks as Directed Graphs	37
4.1	Useful Definitions for Analyzing Predictions (Prediction_Utils)	37
4.2	Desirable Properties of Neural Networks Predictions (Properties)	45
4.2.1	Approximate Comparison of Results	45
4.2.2	Maximum Classifiers	47
4.2.3	Distance-based Properties	47
4.3	Neural Networks as Graphs (NN_Digraph)	54
4.3.1	Neurons as Vertices	56
4.3.2	Arcs (Edges)	57
4.3.3	Updating Neurons	59
4.3.4	Updating arcs (edges)	60
4.3.5	The empty neural network	64
4.3.6	Computing Predictions of Neural Networks	65
4.4	Main Theory (Digraph) (NN_Digraph_Main)	66
5	Neural Networks as Layers	69
5.1	Preliminaries	69
5.1.1	Useful Definitions for Analysing Matrix Predictions (Prediction_Utils_Matrix)	69
5.1.2	Desirable Properties of Neural Networks Predictions (Properties_Matrix)	74
5.1.3	Sequential Layers (NN_Layers)	76

5.1.4	Neural Network Lipschitz Continuity	78
5.2	Models	134
5.2.1	Digraphs as Layers (📄NN_Digraph_Layers)	134
5.2.2	Neural Network as Sequential Layers using Lists (📄NN_Layers_List_Main)	143
5.2.3	Neural Network as Sequential Layers using Vector Spaces (📄NN_Layers_Matrix_Main)	155
5.3	Main Theory (Layers) (📄NN_Layers_Main)	164
5.3.1	Converting between List-based and Matrix-based Sequential Layer Models	165
5.3.2	Converting Between List/Matrix-based Representations Preserves Consistency	165
5.3.3	Semantic Equivalence of List-based and Matrix-based Models	176
6	Main Theory Including all Model Types (📄NN_Main)	185
7	Reference Manual (thy)	187
7.1	Importing Neural Networks and Data (📄NN_Manual)	187
7.2	Proof Methods (📄NN_Manual)	188
8	Examples	189
8.1	Compass	189
8.1.1	Neural Networks as Directed Graphs (📄Compass_Digraph)	189
8.1.2	Neural Networks as List of Layers using List Types (📄Compass_Layers_List)	193
8.1.3	Neural Networks as List of Layers using Matrix Types (📄Compass_Layers_Matrix)	196
8.2	Line Classification Model (📄Grid_Layers) (📄Grid_Layers)	200
8.2.1	Layer-based Modelling using List Types(📄Grid_Layers_List)	201
8.2.2	Layer-based Modelling using List Types (📄Grid_Layers_Matrix)	203

1 Introduction

Machine learning (ML) and, in particular, deep learning (DL) is used successfully in many application areas. Still, their use in safety-critical or security-critical applications is limited, due to the lack of testing and verification techniques that satisfy the stringent requirements of industrial certification standards such as BS EN 50128 [6] (safety) or Common Criteria [7] (security) that are required in such applications. On their highest assurance level, these certification standards require a formal (mathematical) specification of the system, allowing for a formal verification of the system. Moreover, requirements need to be traceable from their elicitation to the execution of test cases on the level of the implementation.

As of today, tools and techniques for certifying high-assurance systems rely on the existence of human-readable program code that can be analyzed, verified, and tested. For systems that are relying on a trained neural network, such a human-readable representation does not exist.

We address this problem by formalizing an important class of neural networks, feed-forward neural networks, in Isabelle/HOL. We present two different approaches of formalizing feed-forward networks and show their equivalence as well as demonstrate their use in verifying certain safety and correctness properties of various example. Moreover, we do not only provide a formal model that allows to reason over feed-forward neural networks, we also provide a datatype package for Isabelle/HOL that supports importing models from TensorFlow.js.

In more detail, our contributions are:

- Two different formal models of feed-forward neural networks in Isabelle/HOL:
 - The first model (see Chapter 4) is based on direct graphs and, hence, is very close to the representation of neural networks in textbooks, e.g., [2].
 - The second model (see Chapter 5) is based on a structure of layers of nodes that share the same activation function. This model is very close to the representation of modern machine learning frameworks such as TensorFlow [1]. For this model, we formalized two variants:
 - * A version optimised for execution that is based on list operations (Section 5.2.2). This model is, usually, also preferred for the verification of a concrete neural network.
 - * A version that is based on vector and matrix operations (Section 5.2.3), which is more suitable for formal reasoning over the model itself.

Moreover, we formally show the equivalence two layer-based models and show that the digraph model is as expressive as the layer-based models.

- A proof of the semantic equivalence of both models (for the subset of models that can be represented in both models).
- A data type package that supports the automatic encoding of machine learning models trained in TensorFlow into our formal framework in Isabelle/HOL.
- A small case studies demonstrating how our formal framework can be used for the verification of safety and correctness trained neural networks.

The main theories for users of this formalisation are:

- For works that build on the formalisation of neural networks as layers (i.e., following the approach of TensorFlow), where the underlying implementation uses the list data type, the theory `NN_Layers_List_Main` (Section 5.2.2) acts as main entry point. For most practical application that have the aim of verifying properties of neural networks, this is the recommended starting point.
- For works that build on the formalisation of neural networks as layers (i.e., following the approach of TensorFlow), where the underlying implementation uses vector and matrix types, the theory `NN_Layers_Matrix_Main` (Section 5.2.3) acts as main entry point.
- The theory `NN_Layers_List_Main` (Section 5.2.2) encodes the TensorFlow-style layers on top of the model using directed graphs.
- The theory `NN_Layers_Main` (Section 5.3) combines all three layer-based models. This is mainly useful for works that focus on meta-level-reasoning, such as proving the equivalence between models or for developing transformations between the different models.
- For works that build on the formalisation of neural networks as directed graphs, the theory `NN_Digraph_Main` (Section 4.4) acts as main entry point.
- The theory `NN_Main` (Chapter 6) combines all models. This is mainly useful for works that focus on meta-level-reasoning, such as proving the equivalence between models or for developing transformations between the different models.
- The theory `NN_Manual` (Chapter 7) contains a brief description of the top-level Isar commands and proof methods provided by this AFP entry.

The rest of this document is automatically generated from the formalization in Isabelle/HOL, i.e., all content is checked by Isabelle. Overall, the structure of this document follows the theory dependencies (see Figure 1.1). A high-level description of this work is published in the proceedings of the International Conference on Formal Methods (FM 2023) [5]:

A. D. Brucker and A. Stell. Verifying feedforward neural networks for classification in Isabelle/HOL. In M. Chechik, J.-P. Katoen, and M. Leucker, editors, Formal Methods (FM 2023). Lübeck, Germany, 2023. ISBN: 978-3-642-38915-3.

A more detailed description, including the presentation of a verification approach for neural networks and further examples, is published in the following PhD Thesis [15]:

A. Stell. Trustworthy Machine Learning for High-Assurance Systems. PhD Thesis. University of Exeter, UK. 2025.

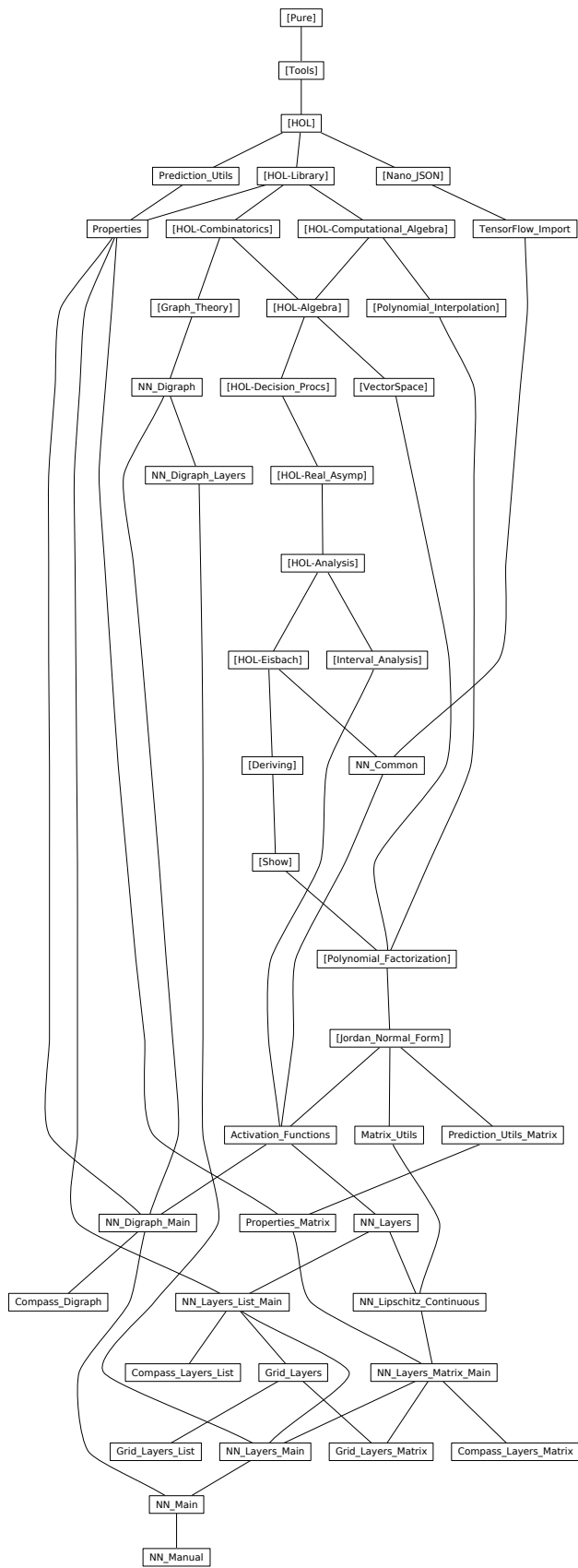


Figure 1.1: The Dependency Graph of the Isabelle Theories.

Generated Sessions

2 Preliminaries

2.1 Proofs and Definitions that Enrich the Matrix Formalization (Matrix_Utils)

```
theory
  Matrix_Utils
imports
  Jordan_Normal_Form.Matrix
  HOL-Combinatorics.Permutations
begin
```

This theory provides additional definition and lemmas that are useful when working with vectors and matrices as provided *Jordan_Normal_Form.Matrix*. Furthermore, this theory contains additional theorems over lists, in particular of properties of *map2* (and, hence, *zip*).

2.1.1 List Properties

```
lemma map2_to_map_idx_eq:
  <length xs = length ys  $\implies$  (map2 (*) xs (ys)) = map ( $\lambda$  i. xs!i * ys!i) [0..< length xs]>
using map2_map_map map_nth
by metis
```

```
lemma map2_to_map_idx:
  <(map2 (*) xs (ys)) = map ( $\lambda$  i. xs!i * ys!i) [0..< min (length xs) (length ys)]>
by (rule nth_equalityI, auto)
```

```
lemma map2_mult_commute:
  <map2 (*) (xs::'a::comm_ring list) ys = map2 (*) ys xs>
by (induction xs ys rule: list_induct2', simp_all add: mult_commute)
```

2.1.2 Vector and Matrix Properties

```
definition mult_vec_mat :: 'a Matrix.vec  $\implies$  'a :: semiring_0 Matrix.mat  $\implies$  'a Matrix.vec (infixl v * 70)
where v v * A  $\equiv$  vec (dim_col A) ( $\lambda$  i. col A i  $\cdot$  v)
```

```
lemma dim_mult_vec_mat: <dim_vec (v v * A) = dim_col A>
by (auto simp add: mult_vec_mat_def)
```

```
lemma index_mult_vec_mat: <i < dim_col A  $\implies$  (v v * A) $ i = col A i  $\cdot$  v>
by (auto simp add: mult_vec_mat_def)
```

```
lemma dim_col_mat_list: < $\forall$  m  $\in$  set (mat_to_list M). dim_col M = length m >
unfolding mat_to_list_def dim_col_def o_def
by simp
```

```
lemma dim_col_mat_list': <mat_to_list M  $\neq$  []  $\implies$  dim_col M = length (hd (mat_to_list M))>
using dim_col_mat_list by fastforce
```

lemma scalar_prod_list:
 $\langle (\text{vec_of_list } v) \cdot (\text{vec_of_list } w) = (\sum i \in \{0 \dots \text{length } w\}. v!i * w!i) \rangle$
unfolding scalar_prod_def vec_of_list_def
by (simp, metis (no_types, lifting) dim_vec sum.cong vec.abs_eq vec_of_list.abs_eq vec_of_list_index)

lemma dim_col_mat_of_col_list: $\langle \text{dim_col } (\text{mat_of_cols_list } n \text{ As}) = \text{length } \text{As} \rangle$
unfolding mat_of_cols_list_def **by** simp

lemma dim_row_mat_of_col_list: $\langle \text{dim_row } (\text{mat_of_cols_list } n \text{ As}) = n \rangle$
unfolding mat_of_cols_list_def **by** simp

lemma dim_col_mat_of_row_list: $\langle \text{dim_col } (\text{mat_of_rows_list } n \text{ As}) = n \rangle$
unfolding mat_of_rows_list_def **by** simp

lemma dim_row_mat_of_row_list: $\langle \text{dim_row } (\text{mat_of_rows_list } n \text{ As}) = \text{length } \text{As} \rangle$
unfolding mat_of_rows_list_def **by** simp

lemma vec_of_list_ext: $\langle \text{vec_of_list } xs = \text{vec_of_list } ys \implies xs = ys \rangle$
by (metis list_vec)

lemma list_of_vec_ext: $\langle \text{list_of_vec } xs = \text{list_of_vec } ys \implies xs = ys \rangle$
by (metis vec_list)

lemma map_if_lam:
 $\langle \text{map } (\lambda i. \text{if } i < n \text{ then } P(i) \text{ else } Q(i)) [0..<n] = \text{map } (\lambda i. P(i)) [0..<n] \rangle$
by simp

lemma map_if_lam':
 $\langle \text{map } (\lambda i. \text{if } p \wedge i < n \text{ then } (P i) \text{ else } (Q i)) [0..<n] = \text{map } (\lambda i. \text{if } p \text{ then } (P i) \text{ else } (Q i)) [0..<n] \rangle$
by simp

lemma map_if_lam'':
 $\langle \text{map } (\lambda i. \text{map } (\lambda ia. \text{if } i < n \text{ then } P i ia \text{ else } Q i ia) [0..<m]) [0..<n] = \text{map } (\lambda i. \text{map } (\lambda ia. P i ia) [0..<m]) [0..<n] \rangle$
by simp

lemma vec_add_list:
assumes $\langle \text{length } v = \text{length } w \rangle$
shows $\langle \text{list_of_vec } ((\text{vec_of_list } v) + (\text{vec_of_list } w)) = \text{map2 } (+) v w \rangle$
unfolding plus_vec_def
apply(simp add:vec_of_list_index)
using assms map2_map_map [of (+)] map_nth
by metis

lemma vec_add_list':
assumes $\langle \text{length } v = \text{length } w \rangle$
shows $\langle ((\text{vec_of_list } v) + (\text{vec_of_list } w)) = \text{vec_of_list } (\text{map2 } (+) v w) \rangle$
apply(rule list_of_vec_ext, simp only: list_vec)
using vec_add_list assms **by** blast

lemma mat_col_list:
assumes $\langle i < \text{length } \text{As} \rangle$
and $\langle \forall a \in \text{set } \text{As}. \forall a' \in \text{set } \text{As}. \text{length } a = \text{length } a' \wedge a \neq [] \rangle$
and $\langle d = \text{length } (\text{hd } \text{As}) \rangle$

```

shows <list_of_vec ( col (mat_of_cols_list d As) i ) = As!i>
apply (intro nth_equalityI)
subgoal using assms by simp
  (metis dim_row_mat_of_col_list list.set_sel(1) list.size(3) not_less_zero nth_mem)
subgoal for j using assms unfolding list_of_vec_index length_list_of_vec
by (auto simp: mat_of_cols_list_def)
done

```

```

lemma mult_vec_mat_col_list:
assumes <length vs=n>
and < $\forall a \in \text{set } As. \forall a' \in \text{set } As. \text{length } a = \text{length } a' \wedge a \neq []$ >
and <length (hd As)=d>
and <length As=n>
and <As  $\neq []$ >
shows <list_of_vec ((vec_of_list vs) v * (mat_of_cols_list d As)) = map ( $\lambda i. \sum ia = 0..<\text{length } vs. As ! i ! ia * vs ! ia$ ) [0..<n]>
```

```

apply (intro nth_equalityI)
subgoal using assms by (simp add: mult_vec_mat_def mat_of_cols_list_def)
subgoal for i using assms unfolding list_of_vec_index mult_vec_mat_def mat_of_cols_list_def
by (auto simp: scalar_prod_def vec_of_list_index intro!: sum.cong arg_cong2[of _ _ _ (*)])
  (metis list.set_sel(1) list_of_vec_index list_of_vec_vec map_nth nth_mem)
done

```

```

lemma mult_vec_mat_row_list:
assumes <length vs=d>
and < $\forall a \in \text{set } As. \forall a' \in \text{set } As. \text{length } a = \text{length } a' \wedge a \neq []$ >
and <length (hd As)=d>
and <length As=n>
and <As  $\neq []$ >
shows <list_of_vec ((vec_of_list vs) v * (mat_of_rows_list d As)) = map ( $\lambda i. \sum ia = 0..<\text{length } vs. \text{map } (\lambda ia. As ! ia ! ia$ ) [0..<length As] ! ia * vs ! ia) [0..<d]>
```

```

apply (intro nth_equalityI)
subgoal using assms by (simp add: mult_vec_mat_def mat_of_rows_list_def)
subgoal for i using assms unfolding list_of_vec_index mult_vec_mat_def mat_of_rows_list_def
by (auto simp: scalar_prod_def vec_of_list_index intro!: sum.cong arg_cong2[of _ _ _ (*)])
  (metis list_of_vec_index list_of_vec_vec)
done

```

```

lemma mult_vec_mat_row_list':
assumes <length vs=d>
and < $\forall a \in \text{set } As. \forall a' \in \text{set } As. \text{length } a = \text{length } a' \wedge a \neq []$ >
and <length (hd As)=d>
and <length As=n>
and <As  $\neq []$ >
shows <((vec_of_list vs) v * (mat_of_rows_list d As)) = vec_of_list (map ( $\lambda i. \sum ia = 0..<\text{length } vs. \text{map } (\lambda ia. As ! ia ! ia$ ) [0..<length As] ! ia * vs ! ia) [0..<d]>
```

```

apply(rule list_of_vec_ext, simp only: list_vec)
using assms mult_vec_mat_row_list by blast

```

```

lemma col_of_rows_list:
assumes <d = Min (set (map length As))>
and <i < d>
shows <list_of_vec (col (mat_of_rows_list d As) i) = map ( $\lambda as. (as!i)$ ) As>
apply (intro nth_equalityI)

```

```

subgoal by (simp add: mat_of_rows_list_def)
subgoal for j using assms unfolding list_of_vec_index
  by (auto simp: mat_of_rows_list_def)
done

```

```

lemma col_of_rows_list':
  assumes  $\langle \forall as \in \text{set } As. \text{length } as = d \rangle$ 
  and  $\langle As \neq [] \rangle$ 
  shows  $\langle \text{col } (\text{mat\_of\_rows\_list } d \ As) \ i = \text{vec\_of\_list } (\text{map } (\lambda as. (as!i)) \ As) \rangle$ 
proof (cases  $i < d$ )
  case True
  then show ?thesis
    apply (subst list_of_vec_ext)
    by (auto simp add: vec_list vec_of_list_ext assms col_of_rows_list)
  next
  case False
  then show ?thesis
  proof –
  have list_of_vec (col (mat_of_rows_list d As) i) =
    list_of_vec (vec_of_list (map ( $\lambda as. as ! i$ ) As))
  proof (rule nth_equalityI)
  show length (list_of_vec (col (mat_of_rows_list d As) i)) =
    length (list_of_vec (vec_of_list (map ( $\lambda as. as ! i$ ) As)))
    by (simp add: mat_of_rows_list_def)
  next
  fix j
  assume j_bound:  $j < \text{length } (\text{list\_of\_vec } (\text{col } (\text{mat\_of\_rows\_list } d \ As) \ i))$ 
  then have j_lt:  $j < \text{length } As$ 
    by (simp add: mat_of_rows_list_def)
  have lhs: list_of_vec (col (mat_of_rows_list d As) i) ! j =
    mat (length As) d ( $\lambda(r, c). As ! r ! c$ ) $$ (j, i)
  unfolding list_of_vec_index mat_of_rows_list_def col_def
  using j_lt by simp
  have rhs: list_of_vec (vec_of_list (map ( $\lambda as. as ! i$ ) As)) ! j =
    (map ( $\lambda as. as ! i$ ) As) ! j
  unfolding list_of_vec_index
  using j_lt vec_of_list_index by blast
  have (map ( $\lambda as. as ! i$ ) As) ! j = As ! j ! i
  using j_lt by simp
  moreover have mat (length As) d ( $\lambda(r, c). As ! r ! c$ ) $$ (j, i) = As ! j ! i
  using j_lt False
  apply (simp add: mat_def assms index_mat_def)
  apply (subst Abs_mat_inverse)
  apply blast
  apply(simp add: mk_mat_def undef_mat_def)
  using assms(1) map_nth nth_mem
  by metis
  ultimately show list_of_vec (col (mat_of_rows_list d As) i) ! j =
    list_of_vec (vec_of_list (map ( $\lambda as. as ! i$ ) As)) ! j
    using lhs rhs by simp
  qed
  then show ?thesis
    by (rule list_of_vec_ext)
qed

```

qed

lemma *list_mat*: $\langle \text{mat_of_rows_list } (\text{dim_col } A) (\text{mat_to_list } A) = A \rangle$
unfolding *mat_to_list_def* *mat_of_rows_list_def*
by (*auto*)

lemma *list_mat_transpose_transpose*: $\langle (\text{mat_of_rows_list } (\text{dim_col } x^T) (\text{mat_to_list } x^T))^T = x \rangle$
using *transpose_transpose* [*of x, symmetric*] *list_mat* **by** *metis*

lemma *mat_list*:
 $\langle \forall r \in \text{set}(rs). \text{length } r = \text{dimc} \implies \text{mat_to_list } (\text{mat_of_rows_list } \text{dimc } rs) = rs \rangle$
unfolding *mat_of_rows_list_def* *mat_to_list_def*
by (*intro nth_equality1, auto*)

lemma *dim_row_list*: $\langle \text{dim_row } m = \text{length } (\text{mat_to_list } m) \rangle$
by (*metis dim_row_mat_of_row_list list_mat*)

lemma *dim_col_list*: $\langle \forall c \in \text{set } (\text{mat_to_list } m). \text{length } c = \text{dim_col } m \rangle$
by (*simp add: mat_to_list_def*)

lemma *scalar_prod_sum_list_lv_eq*:
assumes *same_dim*: $\langle \text{dim_vec } (x::'a::\text{comm_ring Matrix.vec}) = \text{dim_vec } y \rangle$
shows $\langle x \cdot y \equiv \text{sum_list } (\text{map2 } (*) (\text{list_of_vec } x) (\text{list_of_vec } y)) \rangle$
proof (*unfold scalar_prod_def, insert assms, induction dim_vec x*)
case 0
then show ?*case* **by** *simp*
next
case (*Suc xa*) **note** * = *this*
then show ?*case*
apply (*simp add: list_of_vec_map sum_def*)
by (*simp add: comm_monoid_add_class.sum_def interv_sum_list_conv_sum_set_nat map2_map_map*)
qed

lemma *scalar_prod_sum_list_vl_eq*:
assumes *same_dim*: $\langle \text{length } (x::'a::\text{comm_ring list}) = \text{length } y \rangle$
shows $\langle (\text{vec_of_list } x) \cdot (\text{vec_of_list } y) \equiv \text{sum_list } (\text{map2 } (*) x y) \rangle$
proof (*unfold scalar_prod_def, insert assms, induction length x*)
case 0
then show ?*case* **by** *simp*
next
case (*Suc xa*) **note** * = *this*
then show ?*case*
apply (*simp add: list_of_vec_map sum_def*)
using *comm_monoid_add_class.sum_def interv_sum_list_conv_sum_set_nat*
by (*metis (mono_tags, lifting) atLeastLessThan_upt map2_to_map_idx_eq map_eq_conv vec_of_list_index*)
qed

end

2.2 Infrastructure for Importing TensorFlow Models (TensorFlow_Import)

theory

```

TensorFlow_Import
imports
Complex_Main
Nano_JSON.Nano_JSON_Main
keywords
import_TensorFlow :: thy_decl
and as::quasi_command
begin

```

In this theory, we implement the core infrastructure for importing models from TensorFlow.js [14]. This common infrastructure provided a generic parser for the JSON [9, 3] representation of neural networks that can be exported from TensorFlow.js. Actually, TensorFlow.js [14] exports the structure of a neural network (and its configuration used for training the neural network) as JSON file. The weights and biases are stored in a binary file to which the JSON file refers to (see https://www.tensorflow.org/js/guide/save_load and <https://github.com/tensorflow/tfjs/issues/386> for more details).

This theory implements an infrastructure for importing this format, including the decoding of the binary format storing the weights and biases, into Isabelle/HOL. At its core, the infrastructure provides a parser for the format used by TensorFlow.js and a mechanism for hooking datatype packages into it that provide specific encodings into Isabelle/HOL. As a first example, this theory provides a datatype package that provides a JSON-like encoding of neural networks (including their weights and biases) using Nano JSON [4]. The implementation used the JSON infrastructure provided by the AFP entry Nano JSON [4].

2.2.1 Encoder

```

ML<
signature TENSORFLOW_TYPE = sig
  datatype activationT = Linear | BinaryStep | Softsign | Sign | Sigmoid | Swish | Tanh | Relu | Gelu
    | GRelu | Softplus | Elu | Selu | Exponential | Hard_sigmoid
    | Softmax | Softmax_taylor | Sigmoid_taylor
  datatype layerT = Dense | InputLayer | OutputLayer
  type 'a layer = {
    activation: activationT option,
    bias: 'a list, units: int,
    layer_type: layerT,
    name: string,
    weights: 'a list list
  }
end
>
ML_file<Tools/TensorFlow_Type.ML>

```

The ML structure `TensorFlow_Type: TENSORFLOW_TYPE` provides the core datatypes required for the TensorFlow.js import:

- `TensorFlow_Type.activationT`: this datatype enumerates the currently supported activation functions (see Table 3.1 for a mapping of their names used by our formalization).
- `TensorFlow_Type.layerT`: This datatype enumerates the currently supported layer types of TensorFlow.
- `'a TensorFlow_Type.layer`: This record captures the properties of a layer that are extracted from the JSON provided by TensorFlow.js.

```

ML<
signature TENSORFLOW_JSON = sig
  val transform_json: Path.T -> Nano_Json_Type.json -> Nano_Json_Type.json
  val def_nn_json: bstring -> typ -> typ -> Nano_Json_Type.json
    -> local_theory -> local_theory
  val convert_layers: Nano_Json_Type.json
    -> IEEEReal.decimal_approx TensorFlow_Type.layer list
end
>
ML_file<Tools/TensorFlow_Json.ML>

```

TensorFlow.js does export a neural network in a format consisting out of a JSON file and and a binary file:

- the JSON file stores the overall structure of the neural network and the configuration used for training the neural network. Notably, the JSON file does neither contain the biases nor the weights.
- a binary file containing the biases and weights.

The ML structure `TensorFlow_Json:TENSORFLOW_JSON` provides, foremost, a function for parsing the JSON exported neural network in the format supported by TensorFlow. This function, `TensorFlow_Json.transform_json`, takes two arguments

- the directory (path) of the TensorFlow.js export, i.e., the directory in which both the JSON file and the binary file containing the biases and weights are stored.
- the parsed JSON file (the actual JSON parsing is done using `Nano_Json.Parser.json_of_string`, which is provided by Nano JSON [4]).

The function `TensorFlow_Json.transform_json` parses the binary file containing the biases and weights and transforms the input JSON such that the resulting JSON representation includes the biases and weights. In more detail, the JSON file exported by TensorFlow.js stores the biases and weights as follows:

```

"weightsManifest": [{
  "paths": [ "group1-shard1of1.bin" ],
  "weights": [
    {
      "name": "dense/kernel",
      "shape": [2, 1],
      "dtype": "float32"
    }, {
      "name": "dense/bias",
      "shape": [1],
      "dtype": "float32"
    }
  ]
}]

```

JSON

Instead of storing the biases and weights in the JSON file, the exported JSON only contains the type information (here: `float32`) refers to an external file (here: `group1-shard1of1.bin` that stores the actual value. In our example, this external file has a size of 12 bytes, storing three 32 Bit floating point numbers

(encoding as IEEE floating point using a Little Endian encoding. The order of the biases and weights corresponds to the order and shape of their references in the original JSON file. In our example, the function `TensorFlow_Json.transform_json` results in the following transformed `weightsManifest`:

```

"weightsManifest": [{
  "name": "dense/kernel",
  "shape": [
    [-0.47318925857543945E1],
    [-0.4610690593719482E1]
  ],
},
{
  "name": "dense/bias",
  "shape": [[0.22737088203430176E1]]
}]

```

JSON

Moreover, the ML structure `TensorFlow_Json:TENSORFLOW_JSON` also provides an ML for converting a (transformed) JSON representation into a more abstract representation based on a sequence of layers: `TensorFlow_Json.convert_layers`. This function uses the datatypes provided by `TensorFlow_Type:TENSORFLOW_TYPE`.

Finally, `TensorFlow_Json:TENSORFLOW_JSON` provides `TensorFlow_Json.def_nn_json`, which is a simple wrapper around the datatype package provided by Nano JSON generating a formal JSON representation of the neural network imported from `TensorFlow.js` in HOL.

ML

```

signature CONVERT_TENSORFLOW_SYMTAB = sig
  structure NN_Encoder_Data: THEORY_DATA
  eqtype targetT
  type nn_encoderT = bstring -> typ -> typ -> Nano_Json_Type.json -> local_theory -> local_theory
  val add_encoding: Symtab.key * nn_encoderT -> theory -> theory
  val assert_target: theory -> Symtab.key -> Symtab.key
  val lookup_nn_encoder: theory -> Symtab.key -> (targetT * nn_encoderT) option
end

```

>

ML_file<Tools/Convert_TensorFlow_Symtab.ML>

We use the mechanism of attaching a `syntab` to theories to provide a dynamic registration mechanism for different datatype packages that encode the JSON representation in a formal model. The ML structure `Convert_TensorFlow_Symtab:CONVERT_TENSORFLOW_SYMTAB` defines the type for encoder functions (i.e., `Convert_TensorFlow_Symtab.nn_encoderT` and it provides methods for adding a new encoding (`Convert_TensorFlow_Symtab.add_encoding`, checking if an encoder for a given target encoding exists (`Convert_TensorFlow_Symtab.assert_target`, and for the lookup of an encoder (`Convert_TensorFlow_Symtab.lookup_nn_encoder`). The `syntab` is registered as follows:

```

ML<val _ = Theory.setup
  (Convert_TensorFlow_Symtab.add_encoding
   (json, TensorFlow_Json.def_nn_json))

```

>

ML

```

local
  fun import_and_define_nn defN tokenFile modeOption lthy =
    let

```

```

val thy = Proof_Context.theory_of lthy

val ({src_path, lines, ...}:Token.file) = tokenFile
val path = if (Path.is_absolute src_path)
  then src_path
  else Path.append (Resources.master_directory thy) src_path
val mode = case modeOption of
  SOME m => m
  | NONE => json
val encoder = let
  val _ = Convert_TensorFlow_Symtab.assert_target thy mode
  in
  case Convert_TensorFlow_Symtab.lookup_nn_encoder thy mode
  of NONE => error Encoder not found
  | SOME e => snd e
  end
val strT = Nano_Json_Parser.stringT thy
val numT = Nano_Json_Parser.numT thy
val tf_json = Nano_Json_Parser.json_of_string (String.concat lines)
val json = TensorFlow_Json.transform_json (Path.dir path) tf_json
in
  encoder defN strT numT json lthy
end

in
val _ = Outer_Syntax.command command_keyword <import_TensorFlow>
  Import trained neural network from a TensorFlow.js JSON file.
  ((Parse.name -- keyword <file> -- Resources.parse_file -- Scan.option (keyword <as> |-- Parse.!!!
(Parse.name))) >>
  (fn (((name,_),get_file),mode_option) =>
    Toplevel.theory (fn thy => let val file = get_file thy;
  in Named_Target.theory_map (import_and_define_nn name file mode_option) thy end)
  ))
end
>

```

Lastly, we bind our encoder to a new top-level command: **import_TensorFlow** and prepare its default configuration:

```
declare[[JSON_num_type = real, JSON_string_type = string, JSON_verbose = false]]
```

2.2.2 Example Import

In the following, we briefly demonstrate the use of the TensorFlow.js import.

```
import_TensorFlow compass file examples/compass/model/model.json as json
```

```
JSON_export compass file nor_model_transformed
```

This generated the definition *compass* with the following definition:

```
compass ≡
OBJECT
[("format", STRING "layers—model"),
 ("generatedBy", STRING "keras v2.10.0"),
 ("convertedBy", STRING "TensorFlow.js Converter v3.19.0"),
```

```

("modelTopology",
OBJECT
[("keras_version", STRING "2.10.0"),
("backend", STRING "tensorflow"),
("model_config",
OBJECT
[("class_name", STRING "Sequential"),
("config",
OBJECT
[("name", STRING "sequential_1"),
("layers",
ARRAY
[OBJECT
[("class_name", STRING "InputLayer"),
("config",
OBJECT
[("batch_input_shape", ARRAY [NULL, NUMBER 9]),
("dtype", STRING "float32"),
("sparse", BOOL False), ("ragged", BOOL False),
("name", STRING "dense_input")]),
OBJECT
[("class_name", STRING "Dense"),
("config",
OBJECT
[("name", STRING "dense"), ("trainable", BOOL True),
("dtype", STRING "float32"), ("units", NUMBER 3),
("activation", STRING "linear"),
("use_bias", BOOL True),
("kernel_initializer",
OBJECT
[("class_name", STRING "GlorotUniform"),
("config", OBJECT [{"seed", NULL}])]),
("bias_initializer",
OBJECT
[("class_name", STRING "Zeros"),
("config", OBJECT [])]),
("kernel_regularizer", NULL),
("bias_regularizer", NULL),
("activity_regularizer", NULL),
("kernel_constraint", NULL),
("bias_constraint", NULL)]]),
OBJECT
[("class_name", STRING "Dense"),
("config",
OBJECT
[("name", STRING "dense_2"),
("trainable", BOOL True),
("dtype", STRING "float32"), ("units", NUMBER 4),
("activation", STRING "linear"),
("use_bias", BOOL True),
("kernel_initializer",
OBJECT
[("class_name", STRING "Constant"),
("config",

```

```

OBJECT
  [("value",
    ARRAY
      [ARRAY
        [NUMBER
          (4108836501836777 / 10000000000000000),
        NUMBER
          (- 2398796558380127 / 10000000000000000),
        NUMBER
          (- 46464818716049194 / 10000000000000000),
        NUMBER (1946548342704773 / 10000000000000000)],
      ARRAY
        [NUMBER
          (14860405027866364 / 10000000000000000),
        NUMBER
          (- 7789374142885208 / 10000000000000000),
        NUMBER
          (10928256511688232 / 10000000000000000),
        NUMBER
          (- 952406108379364 / 10000000000000000)],
      ARRAY
        [NUMBER
          (24455930292606354 / 10000000000000000),
        NUMBER (5169432163238525 / 10000000000000000),
        NUMBER
          (- 14084954261779785 / 10000000000000000),
        NUMBER
          (- 6348744630813599 /
            10000000000000000)]])]),
("bias_initializer",
  OBJECT
    [("class_name", STRING "Constant"),
    ("config",
      OBJECT
        [("value",
          ARRAY
            [NUMBER (4792080223560333 / 10000000000000000),
            NUMBER
              (- 16364477574825287 / 10000000000000000),
            NUMBER
              (- 24132762849330902 / 10000000000000000),
            NUMBER
              (- 3057991564273834 / 10000000000000000)]])]),
    ("kernel_regularizer", NULL),
    ("bias_regularizer", NULL),
    ("activity_regularizer", NULL),
    ("kernel_constraint", NULL),
    ("bias_constraint", NULL)]])]),
("training_config",
  OBJECT
    [("loss", STRING "categorical_crossentropy"),
    ("metrics",
      ARRAY
        [ARRAY

```

```

[OBJECT
  [{"class_name", STRING "MeanMetricWrapper"},
   ("config",
    OBJECT
      [{"name", STRING "binary_accuracy"},
       ("dtype", STRING "float32"),
       ("fn", STRING "binary_accuracy")]]]),
 ("weighted_metrics", NULL), ("loss_weights", NULL),
 ("optimizer_config",
  OBJECT
    [{"class_name", STRING "Adam"},
     ("config",
      OBJECT
        [{"name", STRING "Adam"},
         ("learning_rate", NUMBER (1 / 1000)), ("decay", NUMBER 0),
         ("beta_1", NUMBER (9 / 10)),
         ("beta_2", NUMBER (999 / 1000)),
         ("epsilon", NUMBER (1 / 10000000)),
         ("amsgrad", BOOL False)]]])]),
 ("weightsManifest",
  ARRAY
    [OBJECT
      [{"name", STRING "dense/kernel"},
       ("shape",
        ARRAY
          [ARRAY
            [NUMBER (6684626 / 10000000), NUMBER (628606 / 10000000),
             NUMBER (9863281 / 100000000)],
          ARRAY
            [NUMBER (- 12952799 / 10000000), NUMBER (3662836 / 10000000),
             NUMBER (9530481 / 10000000)],
          ARRAY
            [NUMBER (- 2857958 / 10000000), NUMBER (6922799 / 10000000),
             NUMBER (35006753 / 100000000)],
          ARRAY
            [NUMBER (17300206 / 10000000), NUMBER (- 37598428 / 100000000),
             NUMBER (7897923 / 10000000)],
          ARRAY
            [NUMBER (63918763 / 100000000), NUMBER (15055849 / 100000000),
             NUMBER (- 58135855 / 100000000)],
          ARRAY
            [NUMBER (- 13919318 / 10000000), NUMBER (10981513 / 10000000),
             NUMBER (5679722 / 10000000)],
          ARRAY
            [NUMBER (- 45270395 / 100000000), NUMBER (17104555 / 1000000000),
             NUMBER (5311743 / 10000000)],
          ARRAY
            [NUMBER (13654941 / 10000000), NUMBER (7420693 / 10000000),
             NUMBER (- 9090567 / 10000000)],
          ARRAY
            [NUMBER (- 18450487 / 100000000), NUMBER (15639223 / 100000000),
             NUMBER (- 4547925 / 10000000)]]]),
      [{"name", STRING "dense/bias"}],

```

```

    ("shape",
    ARRAY
    [ARRAY
    [NUMBER (7082077 / 1000000000), NUMBER (107544795 / 1000000000),
    NUMBER (- 15743796 / 1000000000)]]),
  OBJECT
  [("name", STRING "dense_2/kernel"),
  ("shape",
  ARRAY
  [ARRAY
  [NUMBER (41088365 / 1000000000), NUMBER (- 23987966 / 100000000),
  NUMBER (- 4646482 / 100000000), NUMBER (19465483 / 100000000)],
  ARRAY
  [NUMBER (14860405 / 1000000000), NUMBER (- 7789374 / 1000000000),
  NUMBER (10928257 / 100000000), NUMBER (- 9524061 / 100000000)],
  ARRAY
  [NUMBER (2445593 / 100000000), NUMBER (5169432 / 100000000),
  NUMBER (- 14084954 / 100000000),
  NUMBER (- 63487446 / 100000000)]]),
  OBJECT
  [("name", STRING "dense_2/bias"),
  ("shape",
  ARRAY
  [ARRAY
  [NUMBER (47920802 / 1000000000), NUMBER (- 16364478 / 1000000000),
  NUMBER (- 24132763 / 1000000000),
  NUMBER (- 30579916 / 1000000000)]])])

```

end

2.3 Common Infrastructure (📄 NN_Common)

theory NN_Common

imports

TensorFlow_Import

Complex_Main

HOL-Decision_Procs.Approximation

HOL-Eisbach.Eisbach

keywords

import_data_file :: thy_load

begin

In this theory we define common infrastructure that is used by most formalizations of neural networks.

2.3.1 Utility Functions

ML<

structure nn_common_utils = struct

val mk_Trueprop_eq = Hologic.mk_Trueprop o Hologic.mk_eq

fun define_lemmas name thm_names = Specification.theorems_cmd

```

  [((name, []), map (fn n => (Facts.named n, [])) thm_names)]
  [] false

```

```

fun make_const_def (binding, trm) lthy = let
  val lthy' = snd (Local_Theory.begin_nested lthy)
  val arg = ((binding, NoSyn), ((Thm.def_binding binding, @ {attributes [code]}), trm))
  val (_, (_, thm), lthy'') = Local_Theory.define arg lthy'
in
  (thm, Local_Theory.end_nested lthy'')
end

end
>

```

definition <map_of_list = map_of o (List.enumerate o)>

2.3.2 Data Import

```

ML<
structure Data_Import = struct
  fun parseRealLines lines =
    map ((map (Option.valOf o IEEEReal.fromString)) o (String.tokens Char.isSpace)) lines
    handle Option => (error Invalid data error in parseRealData)
  fun term_of_RealList data =
    let
      fun term_of_real_list reals = HOLogic.mk_list (@ {typ real}) (map (Nano_Json_Type.term_of_real false) reals)
    in
      HOLogic.mk_list (@ {typ real list}) (map term_of_real_list data)
    end
  fun def_data name lines =
    let
      val data_term = term_of_RealList (parseRealLines lines)
    in
      Nano_Json_Parser_Isar.make_const_def (Binding.name name, data_term)
    end
end
end

```

```

val _ = Outer_Syntax.command command_keyword <import_data_file>
  Import test or training data and bind it to a constant.
  ((Resources.parse_file -- keyword <defining> -- Parse.name) >>
  (fn ((get_file, _), name) =>
    Toplevel.theory (fn thy =>
      let
        val ({lines, ...}:Token.file) = get_file thy
        val lines' = List.filter (fn s => (s <> )) lines
      in Named_Target.theory_map (snd o (Data_Import.def_data name lines')) thy end)))
>

```

2.3.3 Common Infrastructure for Proof Tactics

```

ML<

datatype nn_proof_mode = SKIP | SORRY | EVAL | NBE | SAFE
val nn_proof_mode = Attrib.setup_config_string binding <nn_proof_mode> (K nbe)
fun get_nn_proof_mode ctxt =
  (case Config.get ctxt nn_proof_mode of

```

```

skip => SKIP
| sorry => SORRY
| eval => EVAL
| nbe => NBE
| safe => SAFE
| name => error (Bad proof mode: ^ quote name ^ (\skip\, \sorry\, \eval\, \nbe\, or \safe\ expected)))
>

```

ML<

```

structure nn_tactics = struct
  fun prove_simple bname stmt tactic lthy =
    let
      val stmt' = Syntax.check_term lthy stmt
      val thm = Goal.prove lthy [] [] stmt' (fn {context, ...} => tactic context)
        |> Goal.norm_result lthy
        |> Goal.check_finished lthy
    in
      lthy |>
      snd o Local_Theory.note ((bname, []), [thm])
    end

  fun prove_method_simple proof_mode method proof_state =
    let
      fun prove_it method proof_state = Seq.the_result error in proof state (Proof.refine method proof_state)
        |> Proof.global_done_proof
    in
      case proof_mode of
        SKIP => error Error in proof_sate_simple, too late to skip.
      | SORRY => Proof.global_skip_proof true proof_state
      | _ => prove_it method proof_state
    end

  fun normalize_tac ctxt = (CHANGED_PROP o
    (CONVERSION (Nbe.dynamic_conv ctxt)
      THEN_ALL_NEW (TRY o resolve_tac ctxt [True!])))

  fun eval_tac ctxt =
    let val conv = Code_Runtime.dynamic_holds_conv
    in
      CONVERSION (Conv.params_conv ~1 (Conv.concl_conv ~1 o conv) ctxt) THEN'
      resolve_tac ctxt [True!]
    end

  fun eval_or_normalize_tac lthy =
    case get_nn_proof_mode lthy of
      SKIP => error Error: Too late to skip proofs.
    | SORRY => error Error: Too late to sorry proofs.
    | EVAL => eval_tac
    | NBE => normalize_tac
    | SAFE => Code_Simp.dynamic_tac

end

```

>

ML<

```
val nn_verbose = let
  val (nn_verbose_config, nn_verbose_setup) =
    Attrib.config_int (Binding.name nn_verbose) (K o)
in
  Context.>>(Context.map_theory nn_verbose_setup);
  nn_verbose_config
end
```

```
structure nn_log = struct
  fun info _ prfx lthy str state =
    let
      val _ = if (o < (Config.get lthy nn_verbose))
        then warning (prfx^str)
        else ()
    in state end
end
```

>

Finally, we lay out the foundations of our custom proof methods. For this, we utilize Eisbach [12].

named_theorems *nn_layer*

```
method forced_approximation =
  ((approximation 15 | approximation 30 | approximation 60 | approximation 120))
```

```
method predict_layer uses add =
  (simp only: nn_layer add)
```

```
lemmas [nn_layer] = list.map(2) foldl.simps if_False if_True if_cancel if_P if_not_P list.size
  option.simps map.identity Let_def
```

end

3 Activation Functions

```
theory Activation_Functions
imports
  HOL—Analysis.Derivative
  TensorFlow_Import
  NN_Common
  Interval_Analysis.Affine_Functions
  Jordan_Normal_Form.Matrix
begin
```

In this theory, we provide definitions for the most common activation functions. Moreover, we also provide an ML-API for working with HOL-terms of activation functions.

3.1 Defining Activation Functions and Their Derivatives (Activation_Functions)

Many common activation functions use the function $f x = e^x$ (written $f x = \exp x$). For those activation functions, we also define approximations using the Taylor series of the exponential function:

```
definition
  <exp_taylor n x = ( $\sum i = 0..n . x^i / \text{fact } i$ )>

lemma exp_taylor2: exp_taylor 2 (x::real) = (1::real) + x + x^2/2
  by(code_simp, simp)
```

3.1.1 Activation Functions

```
definition
  <identity = ( $\lambda v . v$ )>

lemma identity_linear[simp]: <affine_fun identity>
  unfolding identity_def by simp

definition binary_step :: <a::{zero, ord, one, zero}  $\Rightarrow$  'a> where
  <binary_step = ( $\lambda v . \text{if } v \leq 0 \text{ then } 0 \text{ else } 1$ )>

hide_const sign
definition
  <sign = sgn>

definition
  <softsign = ( $\lambda v . v / (|v| + 1)$ )>
definition
  <logistic L k v_0 = ( $\lambda v . L / (1 + \exp(-k * (v - v_0)))$ )>
definition
  <logistic_taylor n L k v_0 = ( $\lambda v . L / (1 + (\exp_taylor n (-k * (v - v_0))))$ )>
```

definition $\text{sigmoid} :: \text{real} \Rightarrow \text{real}$ where
 $\langle \text{sigmoid} = (\lambda v. 1 / (1 + \exp(-v))) \rangle$

definition
 $\langle \text{sigmoid}_{\text{taylor}} n = (\lambda v. 1 / (1 + (\exp_{\text{taylor}} n (-v)))) \rangle$

lemma $\langle \text{sigmoid} = (\text{logistic} (1.0::\text{real}) 1.0\ 0) \rangle$
unfolding sigmoid_def logistic_def **by** **auto**
lemma $\langle \text{sigmoid}_{\text{taylor}} n = (\text{logistic}_{\text{taylor}} n (1.0::\text{real}) 1.0\ 0) \rangle$
unfolding $\text{sigmoid}_{\text{taylor_def}}$ $\text{logistic}_{\text{taylor_def}}$ **by** **auto**

definition
 $\langle \text{swish} = (\lambda v. v * (\text{sigmoid } v)) \rangle$

definition
 $\langle \text{swish}_{\text{taylor}} n = (\lambda v. v * (\text{sigmoid}_{\text{taylor}} n v)) \rangle$

definition
 $\langle \text{relu} = (\lambda v. \max\ 0\ v) \rangle$

definition
 $\langle \text{generalized_relu } \alpha\ m\ t = (\lambda v. \text{case } m \text{ of Some } m' \Rightarrow \min (\text{if } v \leq t \text{ then } \alpha * v \text{ else } v)\ m' \mid \text{None} \Rightarrow \text{if } v \leq t \text{ then } \alpha * v \text{ else } v) \rangle$

lemma $\langle \text{relu} = (\text{generalized_relu} (0.0::\text{real}) \text{None} (0.0)) \rangle$
unfolding relu_def $\text{generalized_relu_def}$ **by** **auto**

definition
 $\langle \text{softplus} = (\lambda v. \ln (1 + \exp v)) \rangle$

definition
 $\langle \text{elu } \alpha = (\lambda v. \text{if } v \leq 0 \text{ then } \alpha * ((\exp v) - 1) \text{ else } v) \rangle$

definition
 $\langle \text{elu}_{\text{taylor}} n \alpha = (\lambda v. \text{if } v \leq 0 \text{ then } \alpha * ((\exp_{\text{taylor}} n v) - 1) \text{ else } v) \rangle$

definition
 $\langle \text{selu} = (\lambda v. \text{let } \alpha = 1.67326324;$
 $\text{scale} = 1.05070098$
 $\text{in if } v \leq 0 \text{ then } \text{scale} * \alpha * ((\exp v) - 1) \text{ else } \text{scale} * v) \rangle$

definition
 $\langle \text{selu}_{\text{taylor}} n = (\lambda v. \text{let } \alpha = 1.67326324;$
 $\text{scale} = 1.05070098$
 $\text{in if } v \leq 0 \text{ then } \text{scale} * \alpha * ((\exp_{\text{taylor}} n v) - 1) \text{ else } \text{scale} * v) \rangle$

definition
 $\langle \text{prelu } \alpha = (\lambda v. \text{if } v < 0 \text{ then } \alpha * v \text{ else } v) \rangle$

definition
 $\langle \text{silu} = (\lambda v. v / (1 + (\exp (-v)))) \rangle$

definition
 $\langle \text{silu}_{\text{taylor}} n = (\lambda v. v / (1 + (\exp_{\text{taylor}} n (-v)))) \rangle$

definition
 $\langle \text{gaussian} = (\lambda v. \exp (-v^2)) \rangle$

definition
 $\langle \text{gaussian}_{\text{taylor}} n = (\lambda v. \exp_{\text{taylor}} n (-v^2)) \rangle$

definition

$\langle \text{hard_sigmoid} = (\lambda v. \text{if } v < -2.5 \text{ then } 0 \text{ else if } v > 2.5 \text{ then } 1 \text{ else } 0.2 * v + 0.5) \rangle$

definition

$\langle \text{gelu_approx} = (\lambda v. 0.5 * v * (1 + \tanh(\sqrt{2 / \pi}) * (v + 0.044715 * v * v * v))) \rangle$

Note, the error function *erf* is available in the AFP entry [8], which can be used for defining a non-approximated *gelu* activation function.

definition softmax :: ('a::{banach,real_normed_algebra_1,inverse}) list \Rightarrow 'a list **where**

$\langle \text{softmax } vs = \text{map } (\lambda v. \text{exp } v / (\sum v' \leftarrow vs. \text{exp } v')) \text{ vs} \rangle$

definition msoftmax :: ('a::{banach,real_normed_algebra_1,inverse}) vec \Rightarrow 'a vec **where**

$\langle \text{msoftmax } vs = \text{map_vec } (\lambda v. \text{exp } v / (\sum v' \leftarrow (\text{list_of_vec } vs). \text{exp } v')) \text{ vs} \rangle$

definition softmax_{taylor} :: nat \Rightarrow ('a::{banach,real_normed_algebra_1,inverse}) list \Rightarrow 'a list **where**

$\langle \text{softmax}_{\text{taylor}} \ n \ vs = \text{map } (\lambda v. (\text{exp}_{\text{taylor}} \ n \ v) / (\sum v' \leftarrow vs. (\text{exp}_{\text{taylor}} \ n \ v'))) \text{ vs} \rangle$

definition msoftmax_{taylor} :: nat \Rightarrow ('a::{banach,real_normed_algebra_1,inverse}) vec \Rightarrow 'a vec **where**

$\langle \text{msoftmax}_{\text{taylor}} \ n \ vs = \text{map_vec } (\lambda v. (\text{exp}_{\text{taylor}} \ n \ v) / (\sum v' \leftarrow (\text{list_of_vec } vs). (\text{exp}_{\text{taylor}} \ n \ v'))) \text{ vs} \rangle$

lemma softmax_{taylor2}:

$\text{softmax}_{\text{taylor}} \ 2 \ vs = \text{map } (\lambda (v::\text{real}). (1 + v + v^2/2) / (\text{foldl } (+) \ 0 \ (\text{map } (\lambda v'. 1 + v' + v'^2/2) \ vs))) \ vs$

unfolding $\text{softmax}_{\text{taylor_def}} \ \text{exp}_{\text{taylor}} \ 2$

by (*simp* *add*: *Groups.add_ac(2)* *fold_plus_sum_list_rev* *foldl_conv_fold*)

lemma softmax_{taylor2}': $\text{softmax}_{\text{taylor}} \ 2 \ vs = \text{map } (\lambda (v::\text{real}). (1 + v + v^2/2) / (\text{foldl } (\lambda a \ x. a + (1 + x + x^2 / 2)) \ 0 \ vs)) \ vs$

apply (*simp* *add*: $\text{softmax}_{\text{taylor}} \ 2$)

by (*code_simp*, *simp*)

definition

$\langle \text{argmax } vs = \text{map } (\lambda v. \text{if } v = \text{Max } (\text{set } vs) \text{ then } 1 \text{ else } 0) \text{ vs} \rangle$

Table 3.1 provides a mapping from our names of the activation functions to the names used by TensorFlow (see https://www.tensorflow.org/api_docs/python/tf/keras/activations/).

Table 3.1: Mapping of the activation functions supported by TensorFlow.

	TensorFlow 2.8.0	Definition
<i>identity</i>	linear	$identity = (\lambda v. v)$
<i>softsign</i>	softsign	$softsign = (\lambda v. v / (v + 1))$
<i>sigmoid</i>	sigmoid	$sigmoid = (\lambda v. 1 / (1 + \exp(-v)))$
<i>sigmoid_{taylor}</i>	-	$sigmoid_{taylor} ?n = (\lambda v. 1 / (1 + \exp_{taylor} ?n (-v)))$
<i>swish</i>	swish	$swish = (\lambda v. v * sigmoid v)$
<i>swish_{taylor}</i>	-	$swish_{taylor} ?n = (\lambda v. v * sigmoid_{taylor} ?n v)$
<i>tanh</i>	thanh	$tanh ?x = \sinh ?x / \cosh ?x$
<i>generalized_relu</i>	relu	$generalized_relu ?\alpha ?m ?t =$ $(\lambda v. case ?m of None \Rightarrow if v \leq ?t then ?\alpha * v else v \mid Some m' \Rightarrow min (if v \leq ?t then ?\alpha * v else v) m')$
<i>relu</i>	relu (with default parameters)	$relu = max\ 0$
<i>gelu_approx</i>	gelu (approx=True)	$gelu_approx = (\lambda v. 5 / 10 * v * (1 + \tanh(\sqrt{2 / \pi}) * (v + 44715 / 10^6 * v * v * v)))$
-	gelu (approx=False)	-
<i>softplus</i>	softplus	$softplus = (\lambda v. \ln(1 + \exp v))$
<i>elu</i>	elu	$elu ?\alpha = (\lambda v. if v \leq 0 then ?\alpha * (\exp v - 1) else v)$
<i>elu_{taylor}</i>	-	$elu_{taylor} ?n ?\alpha = (\lambda v. if v \leq 0 then ?\alpha * (\exp_{taylor} ?n v - 1) else v)$
<i>selu</i>	selu	$selu =$ $(\lambda v. let \alpha = (167326324::?'a) / (10::?'a)^8; scale = (105070098::?'a) / (10::?'a)^8$ $in if v \leq 0 then scale * \alpha * (\exp v - 1) else scale * v)$
<i>selu_{taylor}</i>	-	$selu_{taylor} ?n =$ $(\lambda v. let \alpha = (167326324::?'a) / (10::?'a)^8; scale = (105070098::?'a) / (10::?'a)^8$ $in if v \leq 0 then scale * \alpha * (\exp_{taylor} ?n v - 1) else scale * v)$
<i>exp</i>	exponential	$exp = (\lambda x. \sum n. x^n /_R fact\ n)$
<i>exp_{taylor}</i>	-	$exp_{taylor} ?n ?x = (\sum i = 0..?n. ?x^i / fact\ i)$
<i>hard_sigmoid</i>	hard_sigmoid	$hard_sigmoid =$ $(\lambda v. if v < -((25::?'a) / (10::?'a)) then 0$ $else if (25::?'a) / (10::?'a) < v then 1 else (2::?'a) / (10::?'a) * v + (5::?'a) / (10::?'a))$
<i>softmax</i>	softmax	$softmax ?vs = map (\lambda v. \exp v / sum_list (map \exp ?vs)) ?vs$
<i>softmax_{taylor}</i>	-	$softmax_{taylor} ?n ?vs = map (\lambda v. \exp_{taylor} ?n v / sum_list (map (\exp_{taylor} ?n) ?vs)) ?vs$

3.1.2 Derivatives of Activation Functions

lemma *has_real_derivative_transform*:

$\langle x \in s \implies (\bigwedge x. x \in s \implies g\ x = f\ x) \implies (f \text{ has_real_derivative } f') \text{ (at } x \text{ within } s) \implies (g \text{ has_real_derivative } f') \text{ (at } x \text{ within } s) \rangle$

by (*simp add: has_derivative_transform has_field_derivative_def*)

lemma *one_plus_exp_eq*: $(1 + \exp v) = (\exp v) * (1 + \exp (-v))$

by (*simp add: distrib_left exp_minus_inverse*)

definition *identity'* = $(\lambda v. 1.0)$

lemma *identity'[simp]*: $\langle (\text{identity has_real_derivative } (\text{identity}' v)) \text{ (at } v) \rangle$

by(*simp add: identity_def identity'_def*)

definition $\langle \text{logistic}' L\ k\ v_0 = (\lambda v. (\exp((-k)*(v-v_0)) * k * L) / (1 + \exp((-k)*(v-v_0)))^2) \rangle$

lemma *logistic'[simp]*: $\langle ((\text{logistic } L\ k\ v_0) \text{ has_real_derivative } ((\text{logistic}' L\ k\ v_0) v)) \text{ (at } v) \rangle$

apply (*simp add: logistic_def logistic'_def, intro derivative_eq_intros, simp_all*)

subgoal by (*metis add_eq_0_iff exp_ge_zero le_minus_one_simps(3)*)

subgoal by (*simp add: power2_eq_square*)

done

definition $\langle \text{tanh}' = (\lambda v. 1 - ((\text{tanh } v)^2)) \rangle$

lemma *tanh'[simp]*: $\langle (\text{tanh has_real_derivative } (\text{tanh}' v)) \text{ (at } v) \rangle$

by (*auto intro: derivative_eq_intros simp add: tanh'_def*)

definition $\langle \text{softplus}' = (\lambda v. 1 / (1 + \exp(-v))) \rangle$

lemma *softplus'[simp]*: $\langle (\text{softplus has_real_derivative } (\text{softplus}' v)) \text{ (at } v) \rangle$

apply (*simp add: softplus_def softplus'_def, intro derivative_eq_intros, simp_all add: add_pos_pos*)

by (*metis one_plus_exp_eq add.left_neutral exp_not_eq_zero mult.right_neutral*

nonzero_divide_mult_cancel_left)

definition $\langle \text{prelu1}' = (\lambda v. 1) \rangle$

lemma *prelu1'[simp]*: $\langle ((\text{prelu } 1) \text{ has_real_derivative } (\text{prelu1}' v)) \text{ (at } v) \rangle$

proof (-)

have *: $\langle (\lambda v. \text{if } v < (0::\text{real}) \text{ then } (1::\text{real}) * v \text{ else } v) = (\lambda v. v) \rangle$ **by** *auto*

show ?thesis

by (*simp add: prelu_def prelu1'_def if_split **)

qed

definition $\langle \text{silu}' = (\lambda v. (1 + \exp(-v) + v * (\exp(-v))) / ((1 + \exp(-v))^2)) \rangle$

lemma *silu'[simp]*: $\langle (\text{silu has_real_derivative } (\text{silu}' v)) \text{ (at } v) \rangle$

apply(*simp add: silu_def silu'_def*)

apply (*intro derivative_eq_intros, simp_all*)

subgoal by (*metis add_eq_0_iff exp_ge_zero le_minus_one_simps(3)*)

subgoal by (*simp add: power2_eq_square*)

done

definition $\langle \text{gaussian}' = (\lambda v. -2 * v * \exp(-v^2)) \rangle$

lemma *gaussian'[simp]*: $\langle (\text{gaussian has_real_derivative } (\text{gaussian}' v)) \text{ (at } v) \rangle$

by (*simp add: gaussian_def gaussian'_def, intro derivative_eq_intros, simp_all, simp*)

3.1.3 Single Class Folding Activation Functions

datatype *activation_{single}* = *Identity* | *Sign* | *BinaryStep* | *Logistic real real real* | *Logistic_{taylor} nat real real real*

```

| Tanh | Sigmoid | Sigmoidtaylor nat | ReLU | GReLU real ⟨real option⟩ real
| Softplus | SoftSign | Swish | Swishtaylor nat | GeLUapprox | ELU real
| ELUtaylor nat real | SELU | SELUtaylor nat | PReLU real | SiLU | SiLUtaylor nat
| Gaussian | Gaussiantaylor nat | Exp | Exptaylor nat | HardSigmoid

```

```

fun  $\varphi_{single}$ :: ⟨activationsingle ⇒ (real ⇒ real) option⟩ where
| ⟨ $\varphi_{single}$  Identity      = Some identity⟩
| ⟨ $\varphi_{single}$  Sign       = Some sign⟩
| ⟨ $\varphi_{single}$  BinaryStep = Some binary_step⟩
| ⟨ $\varphi_{single}$  SoftSign   = Some softsign⟩
| ⟨ $\varphi_{single}$  (Logistic L k v0) = Some (logistic L k v0)⟩
| ⟨ $\varphi_{single}$  (Logistictaylor n L k v0) = Some (logistictaylor n L k v0)⟩
| ⟨ $\varphi_{single}$  Sigmoid    = Some sigmoid⟩
| ⟨ $\varphi_{single}$  (Sigmoidtaylor n) = Some (sigmoidtaylor n)⟩
| ⟨ $\varphi_{single}$  Swish      = Some swish⟩
| ⟨ $\varphi_{single}$  (Swishtaylor n) = Some (swishtaylor n)⟩
| ⟨ $\varphi_{single}$  Tanh      = Some tanh⟩
| ⟨ $\varphi_{single}$  ReLU      = Some relu⟩
| ⟨ $\varphi_{single}$  GeLUapprox = Some gelu_approx⟩
| ⟨ $\varphi_{single}$  (GReLU  $\alpha$  m t) = Some (generalized_relu  $\alpha$  m t)⟩
| ⟨ $\varphi_{single}$  Softplus   = Some softplus⟩
| ⟨ $\varphi_{single}$  (ELU  $\alpha$ ) = Some (elu  $\alpha$ )⟩
| ⟨ $\varphi_{single}$  (ELUtaylor n  $\alpha$ ) = Some (elutaylor n  $\alpha$ )⟩
| ⟨ $\varphi_{single}$  SELU      = Some selu⟩
| ⟨ $\varphi_{single}$  (SELUtaylor n) = Some (selutaylor n)⟩
| ⟨ $\varphi_{single}$  Exp       = Some exp⟩
| ⟨ $\varphi_{single}$  (Exptaylor n) = Some (exptaylor n)⟩
| ⟨ $\varphi_{single}$  HardSigmoid = Some hard_sigmoid⟩
| ⟨ $\varphi_{single}$  (PReLU  $\alpha$ ) = Some (prelu  $\alpha$ )⟩
| ⟨ $\varphi_{single}$  SiLU      = Some silu⟩
| ⟨ $\varphi_{single}$  (SiLUtaylor n) = Some (silutaylor n)⟩
| ⟨ $\varphi_{single}$  Gaussian   = Some gaussian⟩
| ⟨ $\varphi_{single}$  (Gaussiantaylor n) = Some (gaussiantaylor n)⟩

```

The datatype `activationsingle` enumerates a list of standard activation functions that are commonly used as part of computing the weighted sum (fold) of all inputs of a neuron. The function `φ_{single}` provides easy access to the activation function itself.

```

fun  $\varphi_{single}'$ :: ⟨activationsingle ⇒ (real ⇒ real option)⟩ where
| ⟨ $\varphi_{single}'$  Identity    = ( $\lambda v$ . Some (identity' v))⟩
| ⟨ $\varphi_{single}'$  Sign      = ( $\lambda v$ . None)⟩
| ⟨ $\varphi_{single}'$  BinaryStep = ( $\lambda v$ . None)⟩
| ⟨ $\varphi_{single}'$  (Logistic L k v0) = ( $\lambda v$ . Some (logistic' L k v0 v))⟩
| ⟨ $\varphi_{single}'$  (Logistictaylor n L k v0) = ( $\lambda v$ . None)⟩
| ⟨ $\varphi_{single}'$  Tanh      = ( $\lambda v$ . Some (tanh' v))⟩
| ⟨ $\varphi_{single}'$  ReLU      = ( $\lambda v$ . None)⟩
| ⟨ $\varphi_{single}'$  Softplus   = ( $\lambda v$ . Some (softplus' v))⟩
| ⟨ $\varphi_{single}'$  (ELU  $\alpha$ ) = ( $\lambda v$ . None)⟩
| ⟨ $\varphi_{single}'$  (ELUtaylor n  $\alpha$ ) = ( $\lambda v$ . None)⟩
| ⟨ $\varphi_{single}'$  (PReLU  $\alpha$ ) = ( $\lambda v$ . if  $\alpha = 1$  then Some (prelu1' v) else None)⟩
| ⟨ $\varphi_{single}'$  SiLU      = ( $\lambda v$ . Some (silu' v))⟩
| ⟨ $\varphi_{single}'$  (SiLUtaylor n) = ( $\lambda v$ . None)⟩
| ⟨ $\varphi_{single}'$  Gaussian   = ( $\lambda v$ . Some (gaussian' v))⟩
| ⟨ $\varphi_{single}'$  (Gaussiantaylor n) = ( $\lambda v$ . None)⟩
| ⟨ $\varphi_{single}'$  (GReLU v va vb) = ( $\lambda v$ . None)⟩

```

```

|⟨φsingle' GeLUapprox    = (λv. None)⟩
|⟨φsingle' Sigmoid      = (λv. None)⟩
|⟨φsingle' (Sigmoidtaylor n) = (λv. None)⟩
|⟨φsingle' SoftSign     = (λv. None)⟩
|⟨φsingle' Swish        = (λv. None)⟩
|⟨φsingle' (Swishtaylor n) = (λv. None)⟩
|⟨φsingle' SELU        = (λv. None)⟩
|⟨φsingle' (SELUtaylor n) = (λv. None)⟩
|⟨φsingle' Exp         = (λ v. Some (exp v))⟩
|⟨φsingle' (Exptaylor n) = (λ v. None)⟩
|⟨φsingle' HardSigmoid  = (λv. None)⟩

```

The function φ_{single}' defines, for derivable activation functions, their derivative. Note that we require derivability in the mathematical sense. For example, while some machine learning text books consider the binary step function derivable except at the point 0, we consider it non derivable, as the binary step function is non continuous at the point 0. In the following, we also provide the “approximated derivatives” of non-continuous activation functions:

lemma

```

assumes ⟨v ∈ (dom (φsingle' a))⟩
shows  ⟨((λ v. the (φsingle' a) v) has_real_derivative (the (φsingle' a v))) (at v within (dom (φsingle' a)))⟩
using  assms by (cases a, auto)

```

3.1.4 Multiclass Folding Activation Functions

```

datatype activationmulti = mIdentity | mSign | mBinaryStep | mLogistic real real real | mLogistictaylor nat real real
real

```

```

| mTanh | mSigmoid | mSigmoidtaylor nat | mReLU | mGReLU real ⟨real option⟩ real
| mSoftplus | mSoftSign | mSwish | mSwishtaylor nat | mGeLUapprox | mELU real
| mELUtaylor nat real | mSELU | mSELUtaylor nat | mPRELU real | mSiLU | mSiLUtaylor nat
| mGaussian | mGaussiantaylor nat | mExp | mExptaylor nat | mHardSigmoid | mSoftmax
| mSoftmaxtaylor nat | mArgmax

```

```

fun φmulti :: ⟨activationmulti ⇒ (real list ⇒ real list) option⟩ where
|⟨φmulti mIdentity    = Some (map identity)⟩
|⟨φmulti mSign        = Some (map sign)⟩
|⟨φmulti mBinaryStep  = Some (map binary_step)⟩
|⟨φmulti mSoftSign    = Some (map softsign)⟩
|⟨φmulti (mLogistictaylor n L k v0) = Some (map (logistictaylor n L k v0))⟩
|⟨φmulti (mLogistic L k v0) = Some (map (logistic L k v0))⟩
|⟨φmulti mSigmoid     = Some (map sigmoid)⟩
|⟨φmulti (mSigmoidtaylor n) = Some (map (sigmoidtaylor n))⟩
|⟨φmulti mSwish       = Some (map swish)⟩
|⟨φmulti (mSwishtaylor n) = Some (map (swishtaylor n))⟩
|⟨φmulti mTanh        = Some (map tanh)⟩
|⟨φmulti mReLU        = Some (map relu)⟩
|⟨φmulti mGeLUapprox  = Some (map gelu_approx)⟩
|⟨φmulti (mGReLU α m t) = Some (map (generalized_relu α m t))⟩
|⟨φmulti mSoftplus    = Some (map softplus)⟩
|⟨φmulti (mELU α)     = Some (map (elu α))⟩
|⟨φmulti (mELUtaylor n α) = Some (map (elutaylor n α))⟩
|⟨φmulti mSELU        = Some (map selu)⟩
|⟨φmulti (mSELUtaylor n) = Some (map (selutaylor n))⟩
|⟨φmulti mExp         = Some (map exp)⟩
|⟨φmulti (mExptaylor n) = Some (map (exptaylor n))⟩

```

```

|⟨φmulti mHardSigmoid = Some (map hard_sigmoid)⟩
|⟨φmulti (mPRELU α) = Some (map (prelu α))⟩
|⟨φmulti mSiLU = Some (map silu)⟩
|⟨φmulti (mSiLUtaylor n) = Some (map (silutaylor n))⟩
|⟨φmulti mGaussian = Some (map gaussian)⟩
|⟨φmulti (mGaussiantaylor n) = Some (map (gaussiantaylor n))⟩
|⟨φmulti mSoftmax = Some softmax⟩
|⟨φmulti (mSoftmaxtaylor n) = Some (softmaxtaylor n)⟩
|⟨φmulti mArgmax = Some argmax⟩

```

The datatype $\text{activation}_{\text{multi}}$ enumerates a list of standard activation functions that are commonly used as part of computing the weighted sum (fold) of all inputs of a neuron. The function φ_{single} provides easy access to the activation function itself.

3.2 Encoding of Activation Functions (Activation_Functions)

ML<

```

signature ACTIVATION_TERM = sig
  datatype mode = MultiList | MultiMatrix | Single
  datatype activationT = Elu | Exponential | GRelu | Gelu | Hard_sigmoid | Linear | Relu | Selu
    | Sigmoid | Softmax | Softmax_taylor | Softplus | Softsign | Sign | BinaryStep | Swish | Tanh
    | Sigmoid_taylor
  val add_function: binding -> term list -> local_theory -> Proof.context
  val def_phi_tab: mode -> string -> activationT list -> local_theory -> Proof.context
  val term_of_activation_eqn_multi_list: activationT -> term
  val term_of_activation_eqn_multi_matrix: activationT -> term
  val term_of_activation_eqn_single: activationT -> term
  val term_of_activation_multi: activationT -> term
  val term_of_activation_single: activationT -> term
end

```

ML_file <Tools/Activation_Functions.ML>

The ML structure `Activation_Term:ACTIVATION_TERM` provides the core infrastructure to construct HOL terms for the activation on the ML-level.

end

4 Neural Networks as Directed Graphs

4.1 Useful Definitions for Analyzing Predictions (Prediction_Utils)

theory

Prediction_Utils

imports

Complex_Main

begin

Utilities definition $\max_{list} :: \langle 'a::linorder list \Rightarrow 'a \rangle$ where

$\langle \max_{list} = \text{Max o set} \rangle$

definition $\min_{list} :: \langle 'a::linorder list \Rightarrow 'a \rangle$ where

$\langle \min_{list} = \text{Min o set} \rangle$

lemma $\max_{list_is_element}$: $\langle l \neq [] \implies \max_{list} l \in \text{set } l \rangle$

by (metis List.finite_set comp_eq_dest_lhs eq_Max_iff max_list_def set_empty)

lemma $\min_{list_is_element}$: $\langle l \neq [] \implies \min_{list} l \in \text{set } l \rangle$

by (metis List.finite_set comp_eq_dest_lhs eq_Min_iff min_list_def set_empty)

lemma $\max_{list_append_eq}$: $\langle \max_{list} (xs @ [x]) = \max_{list} xs \vee \max_{list} (xs @ [x]) = x \rangle$

proof(cases $\max_{list} (xs @ [x]) = x$)

case True

then show ?thesis by blast

next

case False

then show ?thesis

by (metis (no_types, lifting) List.finite_set Max.subset_imp Max_eq_iff comp_def insertE list.discr list.set(1) list.simps(15) max_list_def max_list_is_element order_antisym rotate1.simps(2) set_rotate1 set_subset_Cons singletonD)

qed

lemma $\min_{list_append_eq}$: $\langle \min_{list} (xs @ [x]) = \min_{list} xs \vee \min_{list} (xs @ [x]) = x \rangle$

proof(cases $\min_{list} (xs @ [x]) = x$)

case True

then show ?thesis by blast

next

case False

then show ?thesis

by (metis (no_types, lifting) List.finite_set Min.subset_imp Min_eq_iff comp_def insertE list.discr list.set(1) list.simps(15) min_list_def min_list_is_element order_antisym rotate1.simps(2) set_rotate1 set_subset_Cons singletonD)

qed

lemma $\max_{list_cons_eq}$: $\langle \max_{list} (x\#xs) = \max_{list} xs \vee \max_{list} (x\#xs) = x \rangle$

proof(cases $\max_{list} (x\#xs) = x$)

case True

then show ?thesis by blast

```

next
case False
then show ?thesis
  by (metis (no_types, lifting) List.finite_set Max.subset_imp Max_eq_iff comp_def insertE
      list.discr list.set(1) list.simps(15) max_list_def max_list_is_element order_antisym
      set_subset_Cons singletonD)
qed
lemma min_list_cons_eq:  $\langle \min_{list} (x\#xs) = \min_{list} xs \vee \min_{list} (x\#xs) = x \rangle$ 
proof(cases min_list (x#xs) = x)
case True
then show ?thesis by blast
next
case False
then show ?thesis
  by (metis (no_types, lifting) List.finite_set Min.subset_imp Min_eq_iff comp_def insertE
      list.discr list.set(1) list.simps(15) min_list_def min_list_is_element order_antisym
      set_subset_Cons singletonD)
qed

lemma max_list_append_limit: assumes  $\langle xs \neq [] \rangle$  shows  $\langle \max_{list} xs \leq \max_{list} (xs @ [x]) \rangle$ 
proof(cases max_list (xs @ [x]) = x)
case True
then show ?thesis using assms
  by (metis List.finite_set Max_ge comp_eq_dest_lhs list.set_intros(2) max_list_def
      max_list_is_element rotate1.simps(2) set_rotate1)
next
case False
then show ?thesis
  by (metis dual_order.refl max_list_append_eq)
qed
lemma min_list_append_limit: assumes  $\langle xs \neq [] \rangle$  shows  $\langle \min_{list} (xs @ [x]) \leq \min_{list} xs \rangle$ 
proof(cases min_list (xs @ [x]) = x)
case True
then show ?thesis using assms
  by (metis List.finite_set Min_antimono comp_eq_dest_lhs min_list_def rotate1.simps(2) set_empty
      set_rotate1 set_subset_Cons)
next
case False
then show ?thesis
  by (metis dual_order.refl min_list_append_eq)
qed

lemma max_list_cons_limit: assumes  $\langle xs \neq [] \rangle$  shows  $\langle \max_{list} xs \leq \max_{list} (x\#xs) \rangle$ 
proof(cases max_list (x#xs) = x)
case True
then show ?thesis using assms
  by (metis List.finite_set Max_ge comp_eq_dest_lhs list.set_intros(2) max_list_def
      max_list_is_element)
next
case False
then show ?thesis
  by (metis dual_order.refl max_list_cons_eq)
qed
lemma min_list_cons_limit: assumes  $\langle xs \neq [] \rangle$  shows  $\langle \min_{list} (x\#xs) \leq \min_{list} xs \rangle$ 

```

```

proof(cases minlist (x#xs) = x)
  case True
  then show ?thesis using assms
  by (metis comp_eq_dest_lhs minlist_append_limit minlist_def rotate1.simps(2) set_rotate1)
next
  case False
  then show ?thesis
  by (metis dual_order.refl minlist_cons_eq)
qed

```

Converting Predictions to Percentages **definition** prediction2percentage :: <real list ⇒ real list> where
 <prediction2percentage l = (let m = max_{list} l in map (λ e. e / m * 100.0) l)>

```

lemma prediction2percentage_is_percentage:
  assumes <0 < maxlist l>
  shows <∀ x ∈ set (prediction2percentage l). x ≤ 100.0>
proof(insert assms, induction l rule: rev_induct)
  case Nil
  then show ?case
  unfolding prediction2percentage_def Let_def maxlist_def o_def
  by(simp)
next
  case (snoc x xs)
  then show ?case
  unfolding prediction2percentage_def Let_def maxlist_def o_def
  by(simp add: divide_le_eq)
qed

```

```

lemma prediction2percentage_id: assumes <maxlist p = 100> shows <prediction2percentage p = p>
unfolding prediction2percentage_def
using assms by(simp)

```

```

lemma prediction2percentage_min_id:
  assumes <0 < maxlist p>
  shows <(o ≤ minlist (prediction2percentage p)) = (o ≤ minlist p)>
proof(insert assms, induction p rule: rev_induct)
  case Nil
  then show ?case
  using assms unfolding prediction2percentage_def Let_def minlist_def maxlist_def o_def
  by(simp)
next
  case (snoc x xs)
  then show ?case
  using assms unfolding prediction2percentage_def Let_def minlist_def maxlist_def o_def
  apply(simp,safe)
  subgoal
  by (metis linorder_not_le not_numeral_le_zero zero_le_divide_iff zero_le_mult_iff)
  subgoal
  by (metis linorder_not_le not_numeral_le_zero zero_le_divide_iff zero_le_mult_iff)
  subgoal by simp
  subgoal by simp
  done
qed

```

Maximum Prediction **definition** `posmax_of :: 'a::linorder list ⇒ (nat × 'a) option` where

`⟨posmax_of l = (let m = maxlist l in find (λ e. snd e = m) (enumerate o l))⟩`

definition `pos_of_max :: 'a::linorder list ⇒ nat option` where

`⟨pos_of_max l = map_option fst (posmax_of l)⟩`

definition `posmax_of' :: 'a::linorder list ⇒ (nat × 'a) option` where

`⟨posmax_of' l = (if l = [] then None else Some ((hd o rev o (sort_key snd) o (enumerate o)) l))⟩`

definition `pos_of_max' :: 'a::linorder list ⇒ nat option` where

`⟨pos_of_max' l = map_option fst (posmax_of' l)⟩`

Minimum Prediction **definition** `posmin_of :: 'a::linorder list ⇒ (nat × 'a) option` where

`⟨posmin_of l = (let m = minlist l in find (λ e. snd e = m) (enumerate o l))⟩`

definition `pos_of_min :: 'a::linorder list ⇒ nat option` where

`⟨pos_of_min l = map_option fst (posmin_of l)⟩`

definition `posmin_of' :: 'a::linorder list ⇒ (nat × 'a) option` where

`⟨posmin_of' l = (if l = [] then None else Some ((hd o rev o (sort_key snd) o (enumerate o)) l))⟩`

definition `pos_of_min' :: 'a::linorder list ⇒ nat option` where

`⟨pos_of_min' l = map_option fst (posmin_of' l)⟩`

lemma `find_append_eq: ⟨find P (xs@[x]) = (if find P xs = None then find P [x] else find P xs)⟩`

proof (`induction xs`)

`case Nil`

`then show ?case by simp`

`next`

`case (Cons a xs)`

`then show ?case by simp`

`qed`

lemma `posmax_of_split: ⟨posmax_of (xs @ [x]) = posmax_of (xs) ∨ posmax_of (xs @ [x]) = Some (length xs,x)⟩`

proof(`cases posmax_of (xs @ [x]) = Some (length xs,x)`)

`case True`

`then show ?thesis by simp`

`next`

`case False note * = this`

`then show ?thesis`

`apply (simp)`

`unfolding posmax_of_def Let_def o_def maxlist_def`

`apply clarsimp`

`apply (simp add: * maxlist_append_eq enumerate_append_eq find_append_eq)`

`apply (cases xs=[], simp)`

`apply (cases find (λ e. snd e = Max (insert x (set xs))) (enumerate o xs) , simp_all)`

`subgoal by (metis max_def)`

`subgoal using comp_def list.simps(15) maxlist_append_eq maxlist_def rotate1.simps(2) set_rotate1`

`by (smt (verit) List.finite_set Max.in_idem enumerate_eq_zip find_None_iff find_cong in_set_conv_nth in_set_zip`

`max_def`

`option.disc1)`

`done`

`qed`

lemma `posmin_of_split: ⟨posmin_of (xs @ [x]) = posmin_of (xs) ∨ posmin_of (xs @ [x]) = Some (length xs,x)⟩`

proof(`cases posmin_of (xs @ [x]) = Some (length xs,x)`)

`case True`

`then show ?thesis by simp`

```

next
case False note * = this
then show ?thesis
apply(simp)
unfolding posmin_of_def Let_def o_def min_list_def
apply (simp add: * min_list_append_eq enumerate_append_eq find_append_eq)
apply (cases xs = [])
apply simp
apply (cases find (λe. snd e = Min (insert x (set xs))) (enumerate o xs), simp_all)
subgoal
by (metis linorder_not_le min.absorb4 min.orderE)
subgoal using
List.finite_set Min.insert Min.insert_remove Min.remove enumerate_eq_zip
find_None_iff find_cong in_set_conv_nth in_set_zip length_append length_map
list.size(3) map_nth min.orderE min_def option.simps(3) set_append set_empty
by (smt (verit) insort_insert_triv set_insort_insert)
done
qed

lemma pos_of_max_split:
⟨pos_of_max (xs @ [x]) = pos_of_max (xs) ∨ pos_of_max (xs @ [x]) = Some (length xs)⟩
using pos_of_max_def posmax_of_split
by (metis fst_eqD option.simps(9))

lemma pos_of_min_split:
⟨pos_of_min (xs @ [x]) = pos_of_min (xs) ∨ pos_of_min (xs @ [x]) = Some (length xs)⟩
using pos_of_min_def posmin_of_split
by (metis fst_eqD option.simps(9))

lemma posmax_of_none: ⟨(posmax_of xs = None) = (xs = [])⟩
proof(induction xs rule:List.rev_induct)
case Nil
then show ?case
by (simp add: posmax_of_def)
next
case (snoc x xs) note * = this
then show ?case
unfolding posmax_of_def Let_def
apply(clarsimp simp add:enumerate_append_eq find_append_eq)
subgoal using max_list_append_eq
by (metis append_is_Nil_conv in_set_conv_decomp_last list.discl max_list_is_element
not_Some_eq old.prod.exhaust self_append_conv2 set_ConsD)
done
qed

lemma posmin_of_none: ⟨(posmin_of xs = None) = (xs = [])⟩
proof(induction xs rule:List.rev_induct)
case Nil
then show ?case
by (simp add: posmin_of_def)
next
case (snoc x xs) note * = this
then show ?case
unfolding posmin_of_def Let_def

```

```

apply(clarsimp simp add:enumerate_append_eq find_append_eq)[1]
using minlist_append_eq
by (metis append_Nil eq_snd_iff in_set_conv_nth list.distinct(1) list.size(3) minlist_is_element not_less_zero option.exhaust set_ConsD)
qed

```

```

lemma posmax_of_in_snd:  $\langle (\text{posmax\_of } xs) = \text{Some } p \implies \text{snd } p \in \text{set } xs \rangle$ 
by (metis (mono_tags, lifting) List.finite_set comp_def eq_Max_iff find_Some_iff maxlist_def option.disc1 posmax_of_def posmax_of_none set_empty)

```

```

lemma posmin_of_in_snd:  $\langle (\text{posmin\_of } xs) = \text{Some } p \implies \text{snd } p \in \text{set } xs \rangle$ 
by (metis (mono_tags, lifting) find_Some_iff minlist_is_element option.disc1 posmin_of_def posmin_of_none)

```

```

lemma pos_of_max_none:  $\langle (\text{pos\_of\_max } xs = \text{None}) = (xs = []) \rangle$ 
by(simp add:pos_of_max_def posmax_of_none)

```

```

lemma pos_of_min_none:  $\langle (\text{pos\_of\_min } xs = \text{None}) = (xs = []) \rangle$ 
by(simp add:pos_of_min_def posmin_of_none)

```

```

lemma take_nth_drop_eq:
assumes  $\langle xs \neq [] \rangle$ 
and  $\langle n < \text{length } xs \rangle$ 
shows  $\langle xs = ((\text{take } n \text{ } xs)@[xs!n]@(\text{drop } (n+1) \text{ } xs)) \rangle$ 
by (simp add: Cons_nth_drop_Suc assms(2))

```

```

lemma max_in:
assumes  $\langle xs \neq [] \rangle$ 
and  $\langle n < \text{length } xs \rangle$ 
and  $\langle \forall x \in \text{set } ((\text{take } n \text{ } xs)@(\text{drop } (n+1) \text{ } xs)). xs!n > x \rangle$ 
shows  $\langle \text{Max } (\text{set } ((\text{take } n \text{ } xs)@[xs!n]@(\text{drop } (n+1) \text{ } xs))) = ((\text{take } n \text{ } xs)@[xs!n]@(\text{drop } (n+1) \text{ } xs))!n \rangle$ 
using assms
apply(simp)
by (metis List.finite_set Max_insert2 id_take_nth_drop order_le_less set_append)

```

```

lemma min_in:
assumes  $\langle xs \neq [] \rangle$ 
and  $\langle n < \text{length } xs \rangle$ 
and  $\langle \forall x \in \text{set } ((\text{take } n \text{ } xs)@(\text{drop } (n+1) \text{ } xs)). xs!n < x \rangle$ 
shows  $\langle \text{Min } (\text{set } ((\text{take } n \text{ } xs)@[xs!n]@(\text{drop } (n+1) \text{ } xs))) = ((\text{take } n \text{ } xs)@[xs!n]@(\text{drop } (n+1) \text{ } xs))!n \rangle$ 
using assms
apply(simp)
by (metis List.finite_set Min_insert2 id_take_nth_drop nless_le set_append)

```

```

lemma max_in':
assumes  $\langle xs \neq [] \rangle$ 
and  $\langle n < \text{length } xs \rangle$ 
and  $\langle \forall x \in \text{set } ((\text{take } n \text{ } xs)@(\text{drop } (n+1) \text{ } xs)). xs!n > x \rangle$ 
shows  $\langle \text{Max } (\text{set } xs) = xs!n \rangle$ 
using assms max_in take_nth_drop_eq by metis

```

```

lemma min_in':
assumes  $\langle xs \neq [] \rangle$ 
and  $\langle n < \text{length } xs \rangle$ 

```

and $\langle \forall x \in \text{set } ((\text{take } n \text{ } xs) @ (\text{drop } (n+1) \text{ } xs)). xs!n < x \rangle$
shows $\langle \text{Min } (\text{set } xs) = xs!n \rangle$
using *assms min_in take_nth_drop_eq by metis*

lemma *snd_numerate_eq*:
 $xs \neq [] \implies n < \text{length } xs \implies j < n \implies \text{snd } (\text{List.enumerate } o \text{ } xs ! j) = xs!j$
by *(simp add: nth_enumerate_eq)*

lemma *nth_lower_max*:
assumes $\langle xs \neq [] \rangle$
and $\langle n < \text{length } xs \rangle$
and $\langle \forall x \in \text{set } ((\text{take } n \text{ } xs) @ (\text{drop } (n+1) \text{ } xs)). x < xs!n \rangle$
shows $\langle \forall j < n. xs!j < xs!n \rangle$
proof –
have $x1: \langle \forall x \in \text{set } ((\text{take } n \text{ } xs)). x < xs!n \rangle$ **using** *assms by simp*
then have $x2: (\forall j < n. (xs!j) \in \text{set } ((\text{take } n \text{ } xs)))$
using *assms(2) in_set_conv_nth by fastforce*
then show *?thesis*
using $x1$ *assms nth_take[of _ n xs] by simp*
qed

lemma *nth_higher_min*:
assumes $\langle xs \neq [] \rangle$
and $\langle n < \text{length } xs \rangle$
and $\langle \forall x \in \text{set } ((\text{take } n \text{ } xs) @ (\text{drop } (n+1) \text{ } xs)). x > xs!n \rangle$
shows $\langle \forall j < n. xs!j > xs!n \rangle$
proof –
have $x1: \langle \forall x \in \text{set } ((\text{take } n \text{ } xs)). x > xs!n \rangle$ **using** *assms by simp*
then have $x2: (\forall j < n. (xs!j) \in \text{set } ((\text{take } n \text{ } xs)))$
using *assms(2) in_set_conv_nth by fastforce*
then show *?thesis*
using $x1$ *assms nth_take[of _ n xs] by simp*
qed

lemma *posmax_of_le*:
assumes $\langle xs \neq [] \rangle$
and $\langle n < \text{length } xs \rangle$
and $\langle \forall x \in \text{set } ((\text{take } n \text{ } xs) @ (\text{drop } (n+1) \text{ } xs)). x < xs!n \rangle$
shows $\langle \text{posmax_of } xs = \text{Some } (n, xs!n) \rangle$
unfolding *posmax_of_def max_list_def o_def Let_def*
apply *(simp add: List.find_Some_iff)*
apply *(rule exI[of _ n])*
apply *(simp add: max_in' assms nth_enumerate_eq)*
apply *(rule conjI)*
subgoal using *assms max_in' by metis*
subgoal
apply *(simp add: assms max_in'[of xs n])*
apply *(simp add: assms snd_numerate_eq[of xs n])*
using *assms nth_lower_max[of xs n] by auto*
done

lemma *posmin_of_le*:
assumes $\langle xs \neq [] \rangle$
and $\langle n < \text{length } xs \rangle$

```

and <∀ x ∈ set ((take n xs)@(drop (n+1) xs)). x > xs!n>
shows <posmin_of xs = Some (n,xs!n)>
unfolding posmin_of_def min_list_def o_def Let_def
apply(simp add: List.find_Some_iff)
apply(rule exI[of _ n])
apply(simp add: min_in' assms nth_enumerate_eq)
apply(rule conjI)
subgoal using assms min_in' by metis
subgoal
  apply(simp add: assms min_in'[of xs n])
  apply(simp add: assms snd_numerate_eq[of xs n])
  using assms nth_higher_min[of xs n] by auto
done

```

```

lemma pos_max_le:
  assumes <xs ≠ []>
  and <n < length xs>
  and <∀ x ∈ set ((take n xs)@(drop (n+1) xs)). x < xs!n>
shows <(pos_of_max xs = Some n)>
  using posmax_of_le[of xs n] assms unfolding pos_of_max_def
  by simp

```

```

lemma pos_min_le:
  assumes <xs ≠ []>
  and <n < length xs>
  and <∀ x ∈ set ((take n xs)@(drop (n+1) xs)). x > xs!n>
shows <(pos_of_min xs = Some n)>
  using posmin_of_le[of xs n] assms unfolding pos_of_min_def
  by simp

```

Distance of Maximum Prediction to Next Highest Prediction definition $\delta_{min} :: \text{real list} \Rightarrow \text{real}$ where
 $\langle \delta_{min} l = (\text{let } m = \max_{list} l \text{ in let } m' = \max_{list} (\text{remove1 } m \text{ l}) \text{ in } |m - m'|) \rangle$

```

lemma leq_linear_real:
  assumes b_bound: <(b::real) ∈ {lb..up}>
  and is_leq_at_bounds: <((c1 * lb + c0 ≤ c1' * lb + c0') ∧ (c1 * up + c0 ≤ c1' * up + c0'))>
shows <c1 * b + c0 ≤ c1' * b + c0'>
proof(cases <lb ≤ up>)
  case True
  then show ?thesis
    using assms
    by auto (smt (verit, best) Groups.mult_ac(2) left_diff_distrib mult_right_mono)
next
  case False
  then show ?thesis using assms by(simp)
qed

```

```

lemma leq_linear_real':
  assumes b_bound: <(b::real) ∈ {lb..up}>
  and is_leq_at_bounds: <((c1 * up + c0 ≤ c1' * up + c0') ∧ (c1 * lb + c0 ≤ c1' * lb + c0'))>
shows <c1 * b + c0 ≤ c1' * b + c0'>
proof(cases <lb ≤ up>)
  case True
  then show ?thesis using assms

```

```

    using leq_linear_real by blast
next
case False
then show ?thesis using assms by(simp)
qed
lemma le_linear_real:
  assumes b_bound:  $\langle b :: \text{real} \rangle \in \{lb..up\}$ 
  and is_leq_at_bounds:  $\langle ((c_1 * lb + c_0 < c_1' * lb + c_0') \wedge (c_1 * up + c_0 < c_1' * up + c_0')) \rangle$ 
shows  $\langle c_1 * b + c_0 < c_1' * b + c_0' \rangle$ 
proof(cases  $\langle lb < up \rangle$ )
case True
then show ?thesis
  using assms
  by auto (smt (verit, best) Groups.mult_ac(2) left_diff_distrib mult_right_mono)
next
case False
then show ?thesis using assms
  by (metis atLeastAtMost_iff nless_le order_le_less_trans)
qed

lemma le_linear_real':
  assumes b_bound:  $\langle b :: \text{real} \rangle \in \{lb..up\}$ 
  and is_leq_at_bounds:  $\langle ((c_1 * up + c_0 < c_1' * up + c_0') \wedge (c_1 * lb + c_0 < c_1' * lb + c_0')) \rangle$ 
shows  $\langle c_1 * b + c_0 < c_1' * b + c_0' \rangle$ 
proof(cases  $\langle lb \leq up \rangle$ )
case True
then show ?thesis using assms
  using le_linear_real by blast
next
case False
then show ?thesis using assms by(simp)
qed

lemma pos_max_leq':  $\langle (\text{pos\_of\_max } xs = \text{Some } n) \implies \forall x \in \text{set } xs. x \leq xs!n \rangle$ 
apply(simp add: pos_of_max_def posmax_of_def max_list_def)
by (smt (verit) List.finite_set Max_ge add_o find_Some_iff fst_conv length_enumerate
nth_enumerate_eq snd_conv)

end

```

4.2 Desirable Properties of Neural Networks Predictions (📖 Properties)

```

theory Properties
imports
  Prediction_Utils
  HOL—Library.Interval
  HOL—Library.Interval_Float
begin

```

4.2.1 Approximate Comparison of Results

```

definition  $\langle \text{approx } a \ \varepsilon \ b = (|a - b| \leq \varepsilon) \rangle$ 
notation approx ( $((\_)/ \approx [\_]) \approx (\_)$ ) [60, 60] 60)

```

fun *checkget_result_list* **where**
 <*checkget_result_list* _ *None* *None* = (*None*,*True*)>
 | <*checkget_result_list* ε (*Some* *xs*) (*Some* *ys*) = (*Some* *xs*, *fold* (\wedge) (*map2* (λ *x y*. $x \approx[\varepsilon] \approx y$) *xs* *ys*) *True*)>
 | <*checkget_result_list* _ *r* _ = (*r*,*False*)>

definition <*check_result_list* $r \in s = \text{snd}(\text{checkget_result_list } \varepsilon r s)$ >
notation *check_result_list* ($((_)/ \approx[(_)] \approx_l (_)) [60, 60] 60$)

fun *checkget_result_singleton* **where**
 <*checkget_result_singleton* _ *None* *None* = (*None*,*True*)>
 | <*checkget_result_singleton* ε (*Some* *x*) (*Some* *y*) = (*Some* *x*, $x \approx[\varepsilon] \approx y$)>
 | <*checkget_result_singleton* _ *r* _ = (*r*,*False*)>

definition <*check_result_singleton* $r \in s = \text{snd}(\text{checkget_result_singleton } \varepsilon r s)$ >
notation *check_result_singleton* ($((_)/ \approx[(_)] \approx_s (_)) [60, 60] 60$)

definition
ensure_testdata_range_list :: <*real* \Rightarrow *real list list* \Rightarrow (*real list* \rightarrow *real list*) \Rightarrow *real list list* \Rightarrow *bool*>
where
 <*ensure_testdata_range_list* *delta* *inputs* *P* *outputs*
 = *foldl* (\wedge) *True*
 (*map* (λ *e*. (*P* (*fst* *e*)) $\approx[\text{delta}] \approx_l \text{Some}(\text{snd } e)$)
 (*zip* *inputs* *outputs*))>
notation *ensure_testdata_range_list* ($(_)\models_l \{(_)\} (_) \{(_)\} [61, 3, 90, 3] 60$)

Interval Arithmetic

definition *interval_distance* :: <'a::preorder, minus, zero, ord> *interval* \Rightarrow 'a *interval* \Rightarrow 'a> **where**
 <*interval_distance* *a b* = (*let* (*la*, *ua*) = *bounds_of_interval* *a*;
 (*lb*, *ub*) = *bounds_of_interval* *b*
 in if $ua \leq lb$ *then* $lb - ua$
else if $ub \leq la$ *then* $la - ub$
else *o*)>

fun *intervals_of_list* **where**
 <*intervals_of_list* _ [] = []>
 | <*intervals_of_list* δ (*x*#*xs*) = (*Interval* ($x - |\delta|$, $x + |\delta|$))#(*intervals_of_list* δ *xs*)>

definition <*intervals_of_l* $\delta = \text{map}(\text{intervals_of_list } \delta)$ >

lemma *interval_in_implies_set*: $(x \in \{a..b\}) \implies (x \in \text{set_of } (\text{Interval } (a,b)))$
by (*metis* *atLeastAtMost_iff* *dual_order.trans* *fst_conv* *lower_Interval* *set_of_eq* *snd_conv* *upper_Interval*)

lemma *in_set_interval*: $a \leq b \implies (x \in \text{set_of } (\text{Interval } (a,b))) = (x \in \{a..b\})$
by (*simp* *add*: *lower_Interval* *set_of_eq* *upper_Interval*)

fun *check_result_list_interval_list* :: <'a::preorder *list option* \Rightarrow 'a *interval list option* \Rightarrow *bool*> **where**
 <*check_result_list_interval_list* *None* *None* = *True*>
 | <*check_result_list_interval_list* (*Some* *xs*) (*Some* *ys*) = *fold* (\wedge) (*map2* (λ *x y*. $x \in \text{set_of } y$) *xs* *ys*) *True*>
 | <*check_result_list_interval_list* _ _ = *False*>

notation *check_result_list_interval_list* ($((_)/ \approx_l (_)) [60, 60] 60$)

We define *check_result_list_interval* for checking that two lists are approximately equal (we need the error interval due to possible rounding errors in IEEE754 arithmetic in python compared to mathematical reals in Isabelle).

definition

```
ensure_testdata_interval_list :: <real list list  $\Rightarrow$  (real list  $\rightarrow$  real list)  $\Rightarrow$  real interval list list  $\Rightarrow$  bool>
where
<ensure_testdata_interval_list inputs P outputs
= foldl ( $\wedge$ ) True
  (map ( $\lambda$  e. let a = (P (fst e)) in let b = Some (snd e) in (a  $\approx_l$  b))
    (zip inputs outputs))>
```

notation *ensure_testdata_interval_list* (\models_{il} $\{(-)\}$ $(-)$ $\{(-)\}$ [3, 90, 3] 60)

Using *check_result_list_interval* we now define the property *ensure_testdata_interval* to check that the (symbolically) computed predictions of a neural network meet our expectations.

4.2.2 Maximum Classifiers

definition

```
ensure_testdata_max_list :: <real list list  $\Rightarrow$  (real list  $\rightarrow$  real list)  $\Rightarrow$  real list list  $\Rightarrow$  bool>
where
<ensure_testdata_max_list inputs P outputs
= foldl ( $\wedge$ ) True
  (map ( $\lambda$  e. case P (fst e) of
    None  $\Rightarrow$  False
    | Some p  $\Rightarrow$  pos_of_max p = pos_of_max (snd e))
    (zip inputs outputs))>
```

notation *ensure_testdata_max_list* (\models_l $\{(-)\}$ $(-)$ $\{(-)\}$ [3, 90, 3] 60)

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, this can be tested using *ensure_testdata_max*

definition *ensure_delta_min* :: <real \Rightarrow (real list \rightarrow real list) \Rightarrow bool> **where**

```
<ensure_delta_min  $\delta$  P = ( $\forall$  xs  $\in$  ran P.  $\delta \leq \delta_{min}$  xs)>
```

notation *ensure_delta_min* $(-)$ \models $(-)$ [61, 90] 60)

lemma *ensure_delta_min_dom*: <ensure_delta_min δ P = (\forall x \in dom P. $\delta \leq \delta_{min}$ (the (P x)))>

by(auto simp add:ensure_delta_min_def dom_def ran_def)

Further properties that we formalised 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) shown in *ensure_delta_min*.

4.2.3 Distance-based Properties

Distance and Measurements

locale *distance* =

```
fixes d::<'a list  $\Rightarrow$  'a list  $\Rightarrow$  ('b::{linordered_ab_group_add})>
```

```
assumes identity: <[[length x = length y]]  $\implies$  (d x y = 0) = (x = y)>
```

```
and symmetry: <(d x y = d y x)>
```

```
and triangle_inequality: <[[length x = length y ; length z = length y]]  $\implies$  (d x z  $\leq$  d x y + d y z)>
```

begin

lemma zero: $\langle (d\ x\ y = 0) = (x = y) \vee (\text{length } x \neq \text{length } y) \rangle$
using *distance_axioms distance_def* **by** *fastforce*

lemma $\langle \llbracket \text{length } x = \text{length } y ; \text{length } z = \text{length } y \rrbracket \implies d\ x\ y + d\ y\ x \geq d\ x\ x \rangle$
using *distance.triangle_inequality distance_axioms* **by** *blast*

lemma $\langle \text{length } x = \text{length } y \implies 0 \leq d\ x\ y \rangle$
using *distance_axioms distance_def*
linordered_ab_group_add_class.zero_le_double_add_iff_zero_le_single_add
by (*metis (mono_tags, opaque_lifting)*)

end

definition *mapfoldr* :: $\langle ('a \Rightarrow 'a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'c \Rightarrow 'c) \Rightarrow 'c \Rightarrow 'a\ \text{list} \Rightarrow 'a\ \text{list} \Rightarrow 'c \rangle$ **where**
 $\langle \text{mapfoldr } \text{map_f } \text{fold_f } e\ \text{xs } \text{ys} = \text{foldr } \text{fold_f } (\text{map2 } (\lambda\ e\ o\ e1 . \text{map_f } e\ o\ e1)\ \text{xs } \text{ys})\ e \rangle$

definition *hamming* :: $\langle 'a\ \text{list} \Rightarrow 'a\ \text{list} \Rightarrow \text{nat} \rangle$ **where**
 $\langle \text{hamming } x\ y = \text{mapfoldr } (=) (\lambda\ e\ a . \text{if } e \text{ then } a \text{ else } a + 1)\ 0\ x\ y \rangle$

lemma *hamming_identity*: $\langle \text{length } x = \text{length } y \implies (\text{hamming } x\ y = 0) = (x = y) \rangle$

proof(*induction rule:list_induct2*)

case *Nil*

then show *?case*

by (*simp add: hamming_def mapfoldr_def*)

next

case (*Cons x xs y ys*) **note** $*$ = *this*

then show *?case*

by (*simp add: hamming_def mapfoldr_def*)

qed

lemma *hamming_symmetry*: $\langle \text{hamming } x\ y = \text{hamming } y\ x \rangle$

apply(*simp add: hamming_def mapfoldr_def*)

using *list_all2_all_nth1* [**where** $P=(=)$, *unfolded list.rel_eq*]

by (*smt (verit, best) map2_map_map map_eq_conv zip_commute zip_map_fst_snd*)

lemma *hamming_unroll*: $\llbracket \text{length } \text{xs} = \text{length } \text{ys} \rrbracket$

$\implies \text{hamming } (x\#\text{xs})\ (y\#\text{ys}) = (\text{if } x = y \text{ then } \text{hamming } \text{xs } \text{ys} \text{ else } 1 + \text{hamming } \text{xs } \text{ys})$

proof(*cases x = y*)

case *True*

then show *?thesis* **by**(*simp add: hamming_def mapfoldr_def*)

next

case *False*

then show *?thesis* **by**(*simp add: hamming_def mapfoldr_def*)

qed

lemma *hamming_triangle_inequality*:

$\langle \llbracket \text{length } \text{xs} = \text{length } \text{ys} ; \text{length } \text{ys} = \text{length } \text{zs} \rrbracket$
 $\implies \text{hamming } \text{xs } \text{zs} \leq \text{hamming } \text{xs } \text{ys} + (\text{hamming } \text{ys } \text{zs}) \rangle$

proof(*induction rule:list_induct3*)

```

case Nil
then show ?case by simp
next
case (Cons x xs y ys z zs)
then show ?case
  by (simp add: hamming_unroll)
qed

```

```

global_interpretation hamming_distance: distance hamming
apply(unfold_locales)
subgoal by(simp add: hamming_identity)
subgoal by(simp add: hamming_symmetry)
subgoal by (metis hamming_triangle_inequality_of_nat_add of_nat_le_iff)
done

```

```

definition manhattan::⟨real list ⇒ real list ⇒ real⟩ where
  ⟨manhattan = mapfoldr (λ x y . |x - y|) (+) 0⟩

```

```

lemma manhattan_unroll: [length xs = length ys]
  ⇒ manhattan (x#xs) (y#ys) = |x - y| + manhattan xs ys
by(simp add: manhattan_def mapfoldr_def)

```

```

lemma manhattan_positive: ⟨length x = length y ⇒ 0 ≤ manhattan x y⟩

```

```

proof(induction rule:list_induct2)
case Nil
then show ?case by (simp add: manhattan_def mapfoldr_def)
next
case (Cons x xs y ys) note * = this
then show ?case
  using manhattan_unroll[of xs ys x y] by simp
qed

```

```

lemma manhattan_identity: ⟨length x = length y ⇒ (manhattan x y = 0) = (x = y)⟩

```

```

proof(induction rule:list_induct2)
case Nil
then show ?case by (simp add: manhattan_def mapfoldr_def)
next
case (Cons x xs y ys) note * = this
then show ?case
  proof(cases x = y)
  case True
  then show ?thesis
  using manhattan_unroll[of xs ys x y]
  by (simp add: Cons.IH Cons.hyps)
  next
  case False
  then show ?thesis
  using manhattan_unroll[of xs ys x y] * manhattan_positive
  by (simp add: add_nonneg_eq_o_iff)
  qed
qed

```

```

lemma manhattan_symmetry: ⟨manhattan x y = manhattan y x⟩

```

```

apply (induct x y rule:list_induct2')
subgoal by(simp add: manhattan_def mapfoldr_def)
subgoal by(simp add: manhattan_def mapfoldr_def)
subgoal by(simp add: manhattan_def mapfoldr_def)
subgoal by(simp add: manhattan_def mapfoldr_def)
done

```

```

lemma manhattan_triangle_inequality:
  <[length xs = length ys ; length ys = length (zs::real list)]
  => manhattan xs zs ≤ manhattan xs ys + (manhattan ys zs)>
proof(induction rule:list_induct3)
  case Nil
  then show ?case by(simp add:manhattan_def mapfoldr_def)
next
  case (Cons x xs y ys z zs)
  then show ?case
    by (simp add: manhattan_unroll)
qed

```

```

global_interpretation manhattan_distance: distance manhattan
apply(unfold_locales)
subgoal by(simp add: manhattan_identity)
subgoal by(simp add: manhattan_symmetry)
subgoal by(simp add: manhattan_triangle_inequality)
done

```

```

definition avg_difference::<real list => real list => real> where
  <avg_difference xs ys = (manhattan xs ys) / (min (length xs) (length ys))>

```

```

global_interpretation avg_difference_distance: distance avg_difference
apply(unfold_locales)
subgoal using avg_difference_def distance_def manhattan_distance.distance_axioms
  by fastforce
subgoal using avg_difference_def distance_def manhattan_distance.distance_axioms
  by (metis min.commute)
subgoal unfolding avg_difference_def distance_def manhattan_distance.distance_axioms
  by (metis add_divide_distrib divide_right_mono manhattan_distance.triangle_inequality of_nat_o_le_iff)
done

```

```

definition euclidean::<real list => real list => real> where
  <euclidean X Y = sqrt (mapfoldr (λ x y . (x-y)2) (+) o X Y)>

```

```

lemma euclidean_positive: <length x = length y => 0 ≤ euclidean x y>
proof(induction rule:list_induct2)
  case Nil
  then show ?case
    by(simp add: euclidean_def mapfoldr_def)
next
  case (Cons x xs y ys)
  then show ?case
    by(simp add: euclidean_def mapfoldr_def)
qed

```

```

lemma euclidean_identity: ⟨length x = length y ⟹ (euclidean x y = 0) = (x = y)⟩
proof(induction rule:list_induct2)
  case Nil
  then show ?case by(simp add: euclidean_def mapfoldr_def)
next
  case (Cons x xs y ys) note * = this
  then show ?case
  proof(cases x = y)
    case True
    then show ?thesis using * by(simp add: euclidean_def mapfoldr_def)
  next
  case False note ** = this
  then have ***: ⟨x ≠ y ⟹ 0 ≠ (x - y)2⟩ by simp
  then show ?thesis apply(simp add: euclidean_def euclidean_positive * ** *** mapfoldr_def)
  using ** euclidean_positive[unfolded euclidean_def mapfoldr_def, simplified]
  by (simp add: Cons.hyps add_nonneg_eq_0_iff)
qed
qed

```

```

lemma euclidean_symmetry: ⟨euclidean x y = euclidean y x⟩
apply (induct x y rule:list_induct2')
subgoal by(simp add: euclidean_def mapfoldr_def)
subgoal by(simp add: euclidean_def mapfoldr_def)
subgoal by(simp add: euclidean_def mapfoldr_def)
subgoal by(simp add: euclidean_def mapfoldr_def power2_commute)
done

```

definition

```

check :: ⟨('a list ⟹ 'a list ⟹ 'b) ⟹ ('b ⟹ bool) ⟹ 'a list ⟹ ('a list → 'a list)
      ⟹ ('a list option ⟹ 'a list option ⟹ bool) ⟹ bool⟩ where
⟨check d P inputref prediction P'
  = (∀ x ∈ dom prediction. P(d inputref x) ⟹ P'(prediction x) (prediction inputref))⟩

```

```

lemma ((∀ l ∈ dom prediction. P(dist i l) ⟹ P'(prediction l) (prediction i)))
  = ((∀ l ∈ {l ∈ dom prediction . P(dist i l)}. P'(prediction l) (prediction i)))
by auto

```

lemma hamming_update_1:

```
length xs = length ys ⟹ hamming xs ys ≤ 1 ⟹ (∃ i. xs = ys[i := xs!i])
```

```
proof(induction rule:list_induct2)
```

```
  case Nil
```

```
  then show ?case by simp
```

```
next
```

```
  case (Cons x' xs' y' ys')
```

```
  then show ?case unfolding hamming_def mapfoldr_def
```

```
  proof(cases x' = y')
```

```
    case True note * = this
```

```
    then have h: hamming xs' ys' ≤ 1 using hamming_unroll by (metis Cons.hyps Cons.prems)
```

```
    then show ?thesis
```

```
    proof —
```

```
      obtain i where xs'_eq: xs' = ys'[i := xs'!i] using Cons.IH h by blast

```

```

show ?thesis
proof (cases i)
  case 0
  then show ?thesis
    apply (intro exl[of_ Suc 0])
    apply simp
    using xs'_eq * by simp
  next
  case (Suc j)
  then show ?thesis
    apply (intro exl[of_ Suc (Suc j)])
    apply (simp split: nat.splits)
    using xs'_eq * Suc by simp
  qed
qed
next
case False note ** = this
then show ?thesis proof(cases hamming xs' ys' = 0)
  case True
  then show ?thesis using hamming_identity
    by (metis Cons.hyps list_update_code(2) nth_Cons_0)
  next
  case False
  then have h: ⟨hamming xs' ys' = 1⟩ using ** Cons.hyps Cons.prems hamming_unroll
    by fastforce
  then show ?thesis
    apply(intro exl[of_ 0])
    apply simp
    using h Cons.hyps Cons.prems hamming_unroll **
    by (metis add_le_same_cancel1 not_one_le_zero)
  qed
qed
qed

```

```

lemma hamming_cases1:
  assumes l: ⟨length xs = length ys⟩
  and h: ⟨hamming xs ys ≤ 1⟩
  and p: ⟨P xs⟩
  and u: ⟨ $\bigwedge i. i < \text{length } xs \wedge ys = xs[i := (ys!i)] \implies P ys$ ⟩
shows ⟨P ys⟩
proof(insert assms, induct xs)
  case Nil
  then show ?case by simp
next
case (Cons x' xs')
then show ?case
  using hamming_update_1[of x'#xs' ys]
    hamming_update_1 One_nat_def linorder_not_le list_update_beyond
  by (metis hamming_symmetry)
qed

```

```

lemma hamming_update_2:
  length xs = length ys  $\implies$  hamming xs ys ≤ 2  $\implies$  ( $\exists i j. xs = (ys[i := xs!i])[j := xs!j]$ )

```

```

proof(induction rule: list_induct2)
  case Nil
  then show ?case by simp
next
  case (Cons x xs y ys)
  then show ?case
    by (metis Suc_1 Suc_eq_plus1_left Suc_le_mono hamming_unroll hamming_update_1
      list_update_code(2) list_update_code(3) nth_Cons_o nth_Cons_Suc)
qed

lemma hamming_cases2:
  assumes l: <length xs = length ys>
  and h: <hamming xs ys ≤ 2>
  and p: <P xs>
  and u: <∧ i j. i < length xs ∧ j < length xs ∧ ys = xs[i := ys!i, j := ys!j] ⇒ P ys>
shows <P ys>
proof(insert assms, induct xs)
  case Nil
  then show ?case by simp
next
  case (Cons x' xs')
  then show ?case
    using hamming_update_2[of x' # xs' ys]
    by (metis hamming_symmetry hamming_update_2 length_list_update list_update_beyond
      list_update_id verit_comp_simplify1(3))
qed

lemma hamming_update_n:
  length xs = length ys ⇒ hamming xs ys = Suc n ⇒ (∃ i. hamming xs (ys[j := xs!i]) = n)
proof(induction rule: list_induct2)
  case Nil
  then show ?case unfolding hamming_def mapfoldr_def by simp
next
  case (Cons x xs y ys)
  then show ?case
  by (metis Suc_eq_plus1_left diff_Suc_1 hamming_unroll length_list_update list_update_code(2)
    list_update_code(3) nth_Cons_o nth_Cons_Suc)
qed

lemma hamming_update_3:
  length xs = length ys ⇒ hamming xs ys ≤ 3 ⇒ (∃ i j k. xs = ys[i := xs!i, j := xs!j, k := xs!k])
proof(induction rule: list_induct2)
  case Nil
  then show ?case by simp
next
  case (Cons x xs y ys)
  then show ?case proof(cases hamming xs ys = 3)
    case True
    then show ?thesis
      using hamming_update_n hamming_update_2
      by (metis (mono_tags, opaque_lifting) Cons.hyps Cons.prem1 One_nat_def Suc_1 le_Suc_eq
        length_Cons length_list_update list_update_id numeral_3_eq_3)
    next
    case False note * = this

```

```

then show ?thesis
proof(cases hamming xs ys < 3)
  case True
    then show ?thesis
    by (metis Cons.hyps One_nat_def Suc_1 hamming_update_2 less_Suc_eq_le list_update_code(2)
      list_update_code(3) nth_Cons_0 nth_Cons_Suc numeral_3_eq_3)
  next
    case False
    then show ?thesis
    using * Cons.prem1 Cons.hyps hamming_unroll[of xs ys x y]
    by(simp split:if_splits)
  qed
qed
qed

lemma hamming_cases3:
  assumes l: <length xs = length ys>
  and h: <hamming xs ys ≤ 3>
  and p: <P xs>
  and u: <∧ i j k. i < length xs ∧ j < length xs ∧ k < length xs ∧ ys = xs[i := ys!i,j := ys!j,k := ys!k] ⇒ P ys>
shows <P ys>
proof(insert assms, induct xs)
  case Nil
    then show ?case by simp
  next
    case (Cons x' xs')
    then show ?case proof(cases hamming (x'#xs') ys = 3)
      case True
        then show ?thesis
        using hamming_symmetry hamming_update_2 hamming_update_3 Cons.prem1 linorder_not_le
          length_list_update length_Cons list_update_id list_update_beyond
        apply(simp)
        by (smt (verit, del_insts) Cons.prem1(2) Cons.prem1(4) hamming_symmetry hamming_update_3
          length_Cons length_list_update linorder_not_le list_update_beyond list_update_id)
      next
        case False
        then have h: hamming (x'#xs') ys ≤ 2 using Cons.prem1 by simp
        then show ?thesis
        using hamming_cases2 Cons.prem1
        by (metis list_update_id)
      qed
    qed
  end

```

4.3 Neural Networks as Graphs (📖 NN_Digraph)

In this theory, we use the AFP entry “Graph Theory” [13] to model neural networks. In particular, we make use of the formalization of directed graphs.

```

theory NN_Digraph
imports
  Graph_Theory.Digraph
begin

```

definition

```
pipe :: <'a => ('a => 'b) => 'b> (infixl <▷> 70) where
  <a ▷ f = f a>
```

We follow the notation used in [2], i.e., a neural network consists our of edges and neurons (nodes).

```
type_synonym id = nat
```

```
record ('a, 'b) Neuron =
```

```
  φ :: 'b      — activation function
  α :: 'a      — learning rate
  β :: 'a      — bias
  uid :: id    — unique identifier
```

```
datatype ('a, 'b) neuron = In id | Out id | Neuron <('a, 'b) Neuron>
```

```
fun uid where
```

```
  <uid (In nid) = nid>
| <uid (Out nid) = nid>
| <uid (Neuron n) = Neuron.uid n>
```

```
record ('a, 'b) edge =
```

```
  ω :: 'a      — weight input to head
  tl :: <('a, 'b) neuron> — source neuron
  hd :: <('a, 'b) neuron> — target neuron
```

```
type_synonym ('a, 'b) nn_pregraph = <((('a, 'b) neuron, ('a, 'b) edge) pre_digraph)>
```

```
definition upd_edge :: <('a, 'b) nn_pregraph => (('a, 'b) edge => ('a, 'b) edge)
  => ('a, 'b) nn_pregraph> where
```

```
  <upd_edge G upd = (
    |
      verts = verts G ,
      arcs = upd ' (arcs G),
      tail = tail G,
      head = head G
    |>
```

```
definition <upd_ω ω' hd_nid tl_nid a = (if uid (hd a) = hd_nid ∧ uid (tl a) = tl_nid
  then (|ω = ω', tl = tl a, hd = hd a |)
  else a)>
```

```
definition upd_neuron :: <('a, 'b) nn_pregraph => (('a, 'b) Neuron => ('a, 'b) Neuron)
  => ('a, 'b) nn_pregraph> where
```

```
  <upd_neuron G upd = (let upd_Neuron = case_neuron In Out (λ n. Neuron (upd n))
    in (
      |
        verts = upd_Neuron ' (verts G) ,
        arcs = (λ a. (| ω = ω a,
          tl = upd_Neuron (tl a),
          hd = upd_Neuron (hd a) |)) ' (arcs G),
        tail = tail G,
        head = head G
      |>>
```

```
definition <upd_φ φ' nid n = (if Neuron.uid n = nid
```

then $(\varphi = \varphi', \alpha = \alpha n, \beta = \beta n, \text{uid} = \text{Neuron.uid } n)$
 else n)

definition $\langle \text{upd}_\beta \beta' n_{id} n = (\text{if } \text{Neuron.uid } n = n_{id}$
 then $(\varphi = \varphi n, \alpha = \alpha n, \beta = \beta', \text{uid} = \text{Neuron.uid } n)$
 else n)

definition $\langle \text{upd}_\alpha \alpha' n_{id} n = (\text{if } \text{Neuron.uid } n = n_{id}$
 then $(\varphi = \varphi n, \alpha = \alpha', \beta = \beta n, \text{uid} = \text{Neuron.uid } n)$
 else n)

A neural network is a directed graph without loops and without multi-edges. Moreover, *id* of neurons are unique.

definition $\text{input_verts} :: \langle ((a, 'b) \text{ neuron}, (a, 'b) \text{ edge}) \text{ pre_digraph} \Rightarrow (a, 'b) \text{ neuron set} \rangle$
where
 $\langle \text{input_verts } G = (\text{verts } G) - (\text{hd } ' \text{ arcs } G) \rangle$

definition $\text{output_verts} :: \langle ((a, 'b) \text{ neuron}, (a, 'b) \text{ edge}) \text{ pre_digraph} \Rightarrow (a, 'b) \text{ neuron set} \rangle$
where
 $\langle \text{output_verts } G = (\text{verts } G) - (\text{tl } ' \text{ arcs } G) \rangle$

definition $\text{internal_verts} :: \langle ((a, 'b) \text{ neuron}, (a, 'b) \text{ edge}) \text{ pre_digraph} \Rightarrow (a, 'b) \text{ neuron set} \rangle$
where
 $\langle \text{internal_verts } G = (\text{verts } G) - ((\text{input_verts } G) \cup (\text{output_verts } G)) \rangle$

locale $\text{nn_pregraph} = \text{digraph } G$
for $G :: \langle (a :: \{\text{comm_monoid_add}, \text{times}, \text{linorder}\}, 'b) \text{ neuron}, (a, 'b) \text{ edge}) \text{ pre_digraph} \rangle +$
assumes $\text{id_vert_inj} : \langle \text{inj_on uid } (\text{verts } G) \rangle$
and $\text{tail_eq_tl} : \langle \text{tail } G = \text{tl} \rangle$
and $\text{head_eq_hd} : \langle \text{head } G = \text{hd} \rangle$
and $\text{ids_growing} : \langle \forall e \in \text{arcs } G. \text{uid } (\text{tl } e) < \text{uid } (\text{hd } e) \rangle$ – Not strictly necessary, but simplifies termination proofs.
begin

lemma $\text{nn_pregraph} : \text{nn_pregraph } G \text{ by intro_locales}$

end

definition $\langle \text{uids } G = \text{uid } ' \text{ verts } G \rangle$

4.3.1 Neurons as Vertices

context nn_pregraph
begin

The operation *add_vert* preserves neural networks

lemma $\text{nn_pregraph_add_neuron} :$
assumes $\langle \text{uid } n \notin (\text{uids } G) \vee n \in \text{verts } G \rangle$
shows $\langle \text{nn_pregraph } (\text{add_vert } n) \rangle$
apply standard
subgoal by $(\text{simp add} : \text{wf_digraph.tail_in_verts wf_digraph_add_vert})$
subgoal by $(\text{simp add} : \text{wf_digraph.head_in_verts wf_digraph_add_vert})$
subgoal by $(\text{simp add} : \text{verts_add_vert})$
subgoal by $(\text{simp add} : \text{arcs_add_vert})$

```

subgoal by (simp add: arcs_add_vert head_add_vert no_loops pre_digraph.tail_add_vert)
subgoal by (simp add: arc_to_ends_def arcs_add_vert head_add_vert no_multi_arcs tail_add_vert)
subgoal proof(cases <n ∈ verts G>)
  case True
  then show ?thesis
  by (simp add: id_vert_inj insert_absorb verts_add_vert)
next
  case False
  then show ?thesis
  using assms verts_add_vert arcs_add_vert head_add_vert no_loops apply(simp)
  using id_vert_inj uids_def image_def by blast
qed
subgoal using tail_eq_tl tail_add_vert by simp
subgoal using head_eq_hd head_add_vert by simp
subgoal using arcs_add_vert ids_growing by blast
done

```

```

definition add_neuron::<('a, 'b) neuron ⇒ ('a, 'b) nn_pregraph> where
  <add_neuron n = (if (uid n ∉ (uids G) ∨ n ∈ verts G) then add_vert n else G)>

```

```

lemma nn_pregraph_add_nn_neuron: <nn_pregraph (add_neuron a)>
  using add_neuron_def nn_pregraph_add_neuron nn_pregraph_axioms
  by simp
end

```

The operation `pre_digraph.del_vert` preserves neural networks

```

context nn_pregraph
begin

```

```

lemma nn_pregraph_del_vert: <nn_pregraph (del_vert n)>
  apply standard
  subgoal by (simp add: wf_digraph.tail_in_verts wf_digraph_del_vert)
  subgoal
  apply(simp add: ends_del_vert no_multi_arcs pre_digraph.arcs_del_vert inj_on_def)
  by (simp add: head_del_vert verts_del_vert)
  subgoal by (simp add: fin_digraph.finite_verts fin_digraph_del_vert)
  subgoal by (simp add: fin_digraph.finite_arcs fin_digraph_del_vert)
  subgoal by (simp add: head_del_vert no_loops pre_digraph.arcs_del_vert tail_del_vert)
  subgoal by (simp add: ends_del_vert no_multi_arcs pre_digraph.arcs_del_vert)
  subgoal
  apply(simp add: ends_del_vert no_multi_arcs pre_digraph.arcs_del_vert inj_on_def)
  by (metis Diff_iff id_vert_inj inj_on_def verts_del_vert)
  subgoal using tail_eq_tl tail_del_vert by simp
  subgoal using head_eq_hd head_del_vert by simp
  subgoal using arcs_del_vert ids_growing by blast
done

```

```

end

```

4.3.2 Arcs (Edges)

```

declare pre_digraph.add_arc_def [code]

```

```

definition <add_nn_edge G a = (if (uid (tl a) ∉ (uids G) ∨ (tl a) ∈ verts G)
  ∧ (uid (hd a) ∉ (uids G) ∨ (hd a) ∈ verts G)
  ∧ uid (hd a) ≠ uid (tl a)
  ∧ ((arc_to_ends G a) ∉ arcs_ends G ∨ a ∈ arcs G)
  ∧ uid (tl a) < uid (hd a)
  then pre_digraph.add_arc G a
  else G)>

```

```

context nn_pregraph
begin

```

The operation `add_arc` preserves neural networks

```

lemma nn_pregraph_add_arc:
  assumes <uid (tl a) ∉ (uids G) ∨ (tl a) ∈ verts G >
  and <uid (hd a) ∉ (uids G) ∨ (hd a) ∈ verts G >
  and <uid (tl a) < uid (hd a)>
  and <uid (hd a) ≠ uid (tl a)>
  and <(arc_to_ends G a) ∉ arcs_ends G ∨ a ∈ arcs G >
shows <nn_pregraph (add_arc a)>
  apply standard
  subgoal by (meson wf_digraph.tail_in_verts wf_digraph_add_arc)
  subgoal by (meson wf_digraph_add_arc wf_digraph_def)
  subgoal by (simp add: pre_digraph.verts_add_arc_conv)
  subgoal by simp
  subgoal using assms
    by (metis head_add_arc head_eq_hd insert_iff no_loops pre_digraph.arcs_add_arc
      pre_digraph.tail_add_arc tail_eq_tl)
  subgoal
    by (metis arc_to_ends_def arcs_add_arc assms(5) insert_iff no_multi_arcs pre_digraph.head_add_arc
      pre_digraph.tail_add_arc wf_digraph.dominates wf_digraph_axioms)
  subgoal using assms
    by (smt (z3) Un_iff head_eq_hd id_vert_inj image_eqI inj_on_def insertE pre_digraph.verts_add_arc_conv
      singletonD tail_eq_tl uids_def)
  subgoal using tail_eq_tl by simp
  subgoal using head_eq_hd by simp
  subgoal by (simp add: assms(3) ids_growing)
done

```

```

declare add_nn_edge_def[code]

```

```

lemma nn_pregraph_add_nn_edge: <nn_pregraph (add_nn_edge G a)>
  using add_nn_edge_def nn_pregraph_add_arc nn_pregraph_axioms
  by metis

```

The operation `del_arc` preserves neural networks

```

lemma nn_pregraph_del_arc: <nn_pregraph (del_arc a)>
  apply standard
  subgoal by simp
  subgoal by simp
  subgoal by simp
  subgoal by simp
  subgoal by (simp add: no_loops)

```

subgoal by (simp add: arc_to_ends_def no_multi_arcs)
subgoal by (simp add: id_vert_inj)
subgoal by (simp add: tail_eq_tl)
subgoal by (simp add: head_eq_hd)
subgoal using tail_eq_tl head_eq_hd ids_growing **by** simp
done

end

4.3.3 Updating Neurons

context nn_pregraph **begin**

lemma upd $_{\varphi}$ _nid_immutable[simp]: $\langle \text{Neuron.uid } n \neq n_{id} \implies n = (\text{upd}_{\varphi} \varphi' n_{id} n) \rangle$
and upd $_{\varphi}$ _id_immutable[simp]: $\langle \text{Neuron.uid } n = \text{Neuron.uid } (\text{upd}_{\varphi} \varphi' n_{id} n) \rangle$
and upd $_{\varphi}$ _ α _immutable[simp]: $\langle \alpha n = \alpha (\text{upd}_{\varphi} \varphi' n_{id} n) \rangle$
and upd $_{\varphi}$ _ β _immutable[simp]: $\langle \beta n = \beta (\text{upd}_{\varphi} \varphi' n_{id} n) \rangle$
and upd $_{\beta}$ _nid_immutable[simp]: $\langle \text{Neuron.uid } n \neq n_{id} \implies n = (\text{upd}_{\beta} \beta' n_{id} n) \rangle$
and upd $_{\beta}$ _id_immutable[simp]: $\langle \text{Neuron.uid } n = \text{Neuron.uid } (\text{upd}_{\beta} \beta' n_{id} n) \rangle$
and upd $_{\beta}$ _ φ _immutable[simp]: $\langle \varphi n = \varphi (\text{upd}_{\beta} \beta' n_{id} n) \rangle$
and upd $_{\beta}$ _ α _immutable[simp]: $\langle \alpha n = \alpha (\text{upd}_{\beta} \beta' n_{id} n) \rangle$
and upd $_{\alpha}$ _nid_immutable[simp]: $\langle \text{Neuron.uid } n \neq n_{id} \implies n = (\text{upd}_{\alpha} \alpha' n_{id} n) \rangle$
and upd $_{\alpha}$ _id_immutable[simp]: $\langle \text{Neuron.uid } n = \text{Neuron.uid } (\text{upd}_{\alpha} \alpha' n_{id} n) \rangle$
and upd $_{\alpha}$ _ φ _immutable[simp]: $\langle \varphi n = \varphi (\text{upd}_{\alpha} \alpha' n_{id} n) \rangle$
and upd $_{\alpha}$ _ β _immutable[simp]: $\langle \beta n = \beta (\text{upd}_{\alpha} \alpha' n_{id} n) \rangle$
by(cases n, simp_all add:upd $_{\varphi}$ _def upd $_{\alpha}$ _def upd $_{\beta}$ _def)+

lemma wf_digraph_update_neuron:

assumes $\langle \forall n. \text{Neuron.uid } n = \text{Neuron.uid } (\text{upd } n) \rangle$
shows $\langle \text{wf_digraph } (\text{upd_neuron } G \text{ upd}) \rangle$
unfolding upd_neuron_def
apply(simp add: wf_digraph assms image_def wf_digraph_def tail_eq_tl head_eq_hd Let_def)
using head_eq_hd head_in_verts tail_eq_tl tail_in_verts
by fastforce

lemma fn_digraph_update_neuron:

assumes $\langle \forall n. \text{Neuron.uid } n = \text{Neuron.uid } (\text{upd } n) \rangle$
shows $\langle \text{fn_digraph } (\text{upd_neuron } G \text{ upd}) \rangle$
apply standard
apply (meson assms wf_digraph.tail_in_verts wf_digraph_update_neuron)
apply (meson assms wf_digraph.head_in_verts wf_digraph_update_neuron)
by(simp_all add: upd_neuron_def Let_def)

lemma nomulti_digraph_update_neuron:

assumes $\langle \forall n. \text{Neuron.uid } n = \text{Neuron.uid } (\text{upd } n) \rangle$
shows $\langle \text{nomulti_digraph } (\text{upd_neuron } G \text{ upd}) \rangle$
apply standard
subgoal by (meson assms wf_digraph_def wf_digraph_update_neuron)
subgoal by (meson upd_neuron_def assms wf_digraph_def wf_digraph_update_neuron)
subgoal
apply(simp add:image_def arc_to_ends_def upd_neuron_def Let_def id_vert_inj assms

```

wf_digraph tail_eq_tl head_eq_hd)
using assms head_eq_hd head_in_verts id_vert_inj inj_onD no_multi_arcs tail_eq_tl tail_in_verts
apply simp
by (smt (verit) arc_to_ends_def edge.select_convs(2,3) inj_onD neuron.simps(10,11,12) uid.elims uid.simps(3))
done

```

lemma *loopfree_digraph_update_neuron*:

```

assumes <math>\forall n. \text{Neuron.uid } n = \text{Neuron.uid } (\text{upd } n)>
shows <math>\langle \text{loopfree\_digraph } (\text{upd\_neuron } G \text{ upd}) \rangle
apply standard
subgoal by (meson assms wf_digraph.tail_in_verts wf_digraph_update_neuron)
subgoal by (meson assms wf_digraph.head_in_verts wf_digraph_update_neuron)
subgoal
  apply (simp add: image_def arc_to_ends_def Let_def upd_neuron_def id_vert_inj assms
    wf_digraph tail_eq_tl head_eq_hd)
  using assms ids_growing order_less_irrefl neuron.distinct(5) neuron.inject(3)
  apply simp
  by (smt (verit) edge.select_convs(2,3) less_not_refl3 neuron.simps(10,11,12) uid.elims uid.simps(3))
done

```

lemma *nn_pregraph_update_neuron*:

```

assumes <math>\forall n. \text{Neuron.uid } n = \text{Neuron.uid } (\text{upd } n)>
shows <math>\langle \text{nn\_pregraph } (\text{upd\_neuron } G \text{ upd}) \rangle
apply standard
subgoal by (meson assms wf_digraph.tail_in_verts wf_digraph_update_neuron)
subgoal by (meson assms wf_digraph.head_in_verts wf_digraph_update_neuron)
subgoal using assms fin_digraph.finite_verts fin_digraph_update_neuron by blast
subgoal using assms fin_digraph.finite_arcs fin_digraph_update_neuron by blast
subgoal by (metis assms(1) loopfree_digraph.no_loops loopfree_digraph_update_neuron)
subgoal by (metis assms(1) nomulti_digraph.no_multi_arcs nomulti_digraph_update_neuron)
subgoal apply (simp add: Let_def assms upd_neuron_def id_vert_inj image_iff inj_on_def uid.elims)
  using assms id_vert_inj inj_on_def neuron.distinct neuron.simps uid.elims by (smt (verit))
subgoal by (simp add: tail_eq_tl upd_neuron_def Let_def)
subgoal by (simp add: head_eq_hd upd_neuron_def Let_def)
subgoal apply (simp add: assms upd_neuron_def ids_growing Let_def) using ids_growing
  by (smt (z3) assms neuron.exhaust neuron.simps(10) neuron.simps(11) neuron.simps(12) uid.simps(3))
done

```

```

lemma nn_pregraph_upd_φ[simp]: <math>\langle \text{nn\_pregraph } (\text{upd\_neuron } G (\text{upd}_\varphi \varphi' n_{id})) \rangle
and nn_pregraph_upd_β[simp]: <math>\langle \text{nn\_pregraph } (\text{upd\_neuron } G (\text{upd}_\beta \beta' n_{id})) \rangle
and nn_pregraph_upd_α[simp]: <math>\langle \text{nn\_pregraph } (\text{upd\_neuron } G (\text{upd}_\alpha \alpha' n_{id})) \rangle
using nn_pregraph_update_neuron by simp_all

```

end

4.3.4 Updating arcs (edges)

context *nn_pregraph* begin

```

lemma upd_ω_tl_immutable[simp]: <math>\langle (tl \ a = tl \ (\text{upd}_\omega \ \omega' \ n_{hd} \ n_{tl} \ a)) \rangle
and upd_ω_hd_immutable[simp]: <math>\langle (hd \ a = hd \ (\text{upd}_\omega \ \omega' \ n_{hd} \ n_{tl} \ a)) \rangle
by (auto simp: upd_ω_def split: if_split)

```

lemma *upd_ω_ends_immutable*[simp]: $\langle \text{arc_to_ends } G \ a = \text{arc_to_ends } G \ (\text{upd}_\omega \ \omega' \ n_{hd} \ n_{tl} \ a) \rangle$
by (auto simp add: arc_to_ends_def head_eq_hd tail_eq_tl)

lemma *upd_edge_tail_immutable*:
 $\langle \text{tail } (\text{upd_edge } G \ \text{upd}) = \text{tail } G \rangle$
by (simp add: upd_edge_def)

lemma *upd_edge_head_immutable*:
 $\langle \text{head } (\text{upd_edge } G \ \text{upd}) = \text{head } G \rangle$
by (simp add: upd_edge_def)

lemma *upd_edge_vert_immutable*: $\langle \text{verts } (\text{upd_edge } G \ \text{upd}) = \text{verts } G \rangle$
by(simp add: upd_edge_def)

lemma *upd_edge_arcs*: $\langle a \in \text{arcs } (\text{upd_edge } G \ \text{upd}) \implies \exists x \in \text{arcs } G. a = \text{upd } x \rangle$
by (auto simp: upd_edge_def)

lemma *wf_digraph_update_edge*:
assumes $\langle \forall a \in \text{arcs } G. (\text{arc_to_ends } G \ a = \text{arc_to_ends } G \ (\text{upd } a)) \rangle$
shows $\langle \text{wf_digraph } (\text{upd_edge } G \ \text{upd}) \rangle$
apply unfold_locales
subgoal
using assms *upd_edge_vert_immutable upd_edge_tail_immutable*
apply(simp add:upd_edge_def arc_to_ends_def image_def)
using tail_in_verts **by** auto
subgoal
using assms *upd_edge_vert_immutable upd_edge_tail_immutable*
apply(simp add:upd_edge_def arc_to_ends_def image_def)
using head_in_verts **by** auto
done

lemma *fin_digraph_update_edge*:
assumes $\langle \forall a \in \text{arcs } G. (\text{arc_to_ends } G \ a = \text{arc_to_ends } G \ (\text{upd } a)) \rangle$
shows $\langle \text{fin_digraph } (\text{upd_edge } G \ \text{upd}) \rangle$
by (metis fin_digraph_axioms fin_digraph_axioms_def fin_digraph_def finite_image1
pre_digraph.select_convs(1) pre_digraph.select_convs(2) upd_edge_def
wf_digraph_update_edge assms)

lemma *nomulti_digraph_update_edge*:
assumes $\langle \forall a \in \text{arcs } G. (\text{arc_to_ends } G \ a = \text{arc_to_ends } G \ (\text{upd } a)) \rangle$
shows $\langle \text{nomulti_digraph } (\text{upd_edge } G \ \text{upd}) \rangle$
apply standard
subgoal using assms **by** (meson wf_digraph_def wf_digraph_update_edge)
subgoal using assms **by** (meson wf_digraph_def wf_digraph_update_edge)
subgoal using assms
by (metis arc_to_ends_def local.upd_edge_arcs nn_pregraph.upd_edge_head_immutable
nn_pregraph.upd_edge_tail_immutable nn_pregraph_axioms no_multi_arcs)
done

lemma *loopfree_digraph_update_edge*:
assumes $\langle \forall a \in \text{arcs } G. (\text{arc_to_ends } G \ a = \text{arc_to_ends } G \ (\text{upd } a)) \rangle$
shows $\langle \text{loopfree_digraph } (\text{upd_edge } G \ \text{upd}) \rangle$

```

apply standard
subgoal using assms by (simp add: wf_digraph.tail_in_verts wf_digraph_update_edge)
subgoal using assms by (simp add: wf_digraph.head_in_verts wf_digraph_update_edge)
subgoal using assms by (metis adj_not_same arc_to_ends_def dominatesI local.upd_edge_arcs
    upd_edge_head_immutable upd_edge_tail_immutable)
done

```

```

lemma nn_pregraph_update_edge:
  assumes  $\langle \forall a \in \text{arcs } G. (\text{arc\_to\_ends } G \ a = \text{arc\_to\_ends } G \ (\text{upd } a)) \rangle$ 
  and  $\langle \forall a \in \text{arcs } G. \text{uid } (\text{tl } (\text{upd } a)) < \text{uid } (\text{hd } (\text{upd } a)) \rangle$ 
shows  $\langle \text{nn\_pregraph } (\text{upd\_edge } G \ \text{upd}) \rangle$ 
apply(simp add: digraph_def fin_digraph_update_edge head_eq_hd id_vert_inj loopfree_digraph_update_edge
  nn_pregraph_axioms_def nn_pregraph_def nomulti_digraph_update_edge
  tail_eq_tl upd_edge_def assms upd_edge_arcs nn_pregraph_axioms.intro
  upd_edge_head_immutable upd_edge_tail_immutable upd_edge_vert_immutable)
by (metis assms(1) fin_digraph_update_edge head_eq_hd loopfree_digraph_update_edge
  nomulti_digraph_update_edge tail_eq_tl upd_edge_def)

```

```

lemma nn_pregraph_upd $\omega$ [simp]:  $\langle \text{nn\_pregraph } (\text{upd\_edge } G \ (\text{upd}_\omega \ \omega' \ n_{hd} \ n_{tl})) \rangle$ 
using nn_pregraph_update_edge
by (metis (no_types, lifting) ids_growing upd $\omega$ _ends_immutable upd $\omega$ _hd_immutable upd $\omega$ _tl_immutable)

```

end

```

record ('a, 'b, 'c) neural_network =
  graph :: (( 'a, 'b) neuron, ('a, 'b) edge) pre_digraph
  activation_tab ::  $\langle 'b \Rightarrow 'c \ \text{option} \rangle$ 

```

```

definition upd_edge' ::  $\langle ('a, 'b, 'c) \ \text{neural\_network} \Rightarrow (('a, 'b) \ \text{edge} \Rightarrow ('a, 'b) \ \text{edge})$ 
   $\Rightarrow ('a, 'b, 'c) \ \text{neural\_network} \rangle$  where
   $\langle \text{upd\_edge}' \ N \ \text{upd} = \{$ 
    graph = upd_edge (graph N) upd,
    activation_tab = activation_tab N
  \}

```

```

definition upd_neuron' ::  $\langle ('a, 'b, 'c) \ \text{neural\_network} \Rightarrow (('a, 'b) \ \text{Neuron} \Rightarrow ('a, 'b) \ \text{Neuron})$ 
   $\Rightarrow ('a, 'b, 'c) \ \text{neural\_network} \rangle$  where
   $\langle \text{upd\_neuron}' \ N \ \text{upd} = \{$ 
    graph = upd_neuron (graph N) upd,
    activation_tab = activation_tab N
  \}

```

```

definition input_layer ::  $\langle ('a, 'b, 'c) \ \text{neural\_network} \Rightarrow ('a, 'b) \ \text{neuron set} \rangle$  where
   $\langle \text{input\_layer} \ N = \text{input\_verts} \ (\text{graph } N) \rangle$ 

```

```

definition arity ::  $\langle ('a, 'b, 'c) \ \text{neural\_network} \Rightarrow \text{nat} \rangle$  where
   $\langle \text{arity} \ N = \text{card} \ (\text{input\_layer } N) \rangle$ 

```

```

definition input_layer_ids ::  $\langle ('a, 'b, 'c) \ \text{neural\_network} \Rightarrow \text{id set} \rangle$  where
   $\langle \text{input\_layer\_ids} \ N = \text{uid } ' \ (\text{input\_layer } N) \rangle$ 

```

```

definition output_layer ::  $\langle ('a, 'b, 'c) \ \text{neural\_network} \Rightarrow ('a, 'b) \ \text{neuron set} \rangle$  where

```

$\langle \text{output_layer } N = \text{output_verts } (\text{graph } N) \rangle$

definition $\text{output_layer_ids} :: \langle ('a, 'b, 'c) \text{neural_network} \Rightarrow \text{id set} \rangle$ **where**
 $\langle \text{output_layer_ids } N = \text{uid } ' (\text{output_layer } N) \rangle$

definition $\text{incoming_arcs} :: \langle ('a, 'b, 'c) \text{neural_network} \Rightarrow \text{id} \Rightarrow ('a, 'b) \text{edge set} \rangle$ **where**

$\langle \text{incoming_arcs } N \ n_{id} = \{a . a \in \text{arcs } (\text{graph } N) \wedge \text{uid } (\text{hd } a) = n_{id}\} \rangle$

definition $\langle \text{sorted_list_of_set}' \equiv \text{map_fun } \text{id } \text{id } (\text{folding_on.F } (\text{insert_key } (\lambda x. \text{uid } (\text{tl } x)))) \rangle$

definition $\text{incoming_arcs_l} :: \langle ('a, 'b, 'c) \text{neural_network} \Rightarrow \text{id} \Rightarrow ('a, 'b) \text{edge list} \rangle$ **where**
 $\langle \text{incoming_arcs_l } N \ n_{id} = \text{sorted_list_of_set}' (\text{incoming_arcs } N \ n_{id}) \rangle$

context nn_pregraph **begin**

lemma $\text{incoming_arcs_l_eq_incoming_arcs} : \langle \text{set } (\text{incoming_arcs_l } N \ n_{id}) = (\text{incoming_arcs } N \ n_{id}) \rangle$
unfolding $\text{incoming_arcs_l_def sorted_list_of_set}'_def$
oops

lemma $\text{incoming_arcs_l_alt_def} : \langle (\text{incoming_arcs_l } N \ n_{id})$
 $= (\text{sorted_key_list_of_set } (\lambda x. \text{uid } (\text{tl } x)) (\text{incoming_arcs } N \ n_{id})) \rangle$
unfolding $\text{incoming_arcs_l_def sorted_list_of_set}'_def \text{incoming_arcs_def}$
by $(\text{simp add: sorted_key_list_of_set_def})$

lemma $\text{insert_key_comm} : \text{inj } f \implies (\text{insert_key } f y \circ \text{insert_key } f x) = (\text{insert_key } f x \circ \text{insert_key } f y)$
apply $(\text{cases } x = y)$
by $(\text{auto intro: insert_key_left_comm simp add: inj_def inj_on_def fun_eq_iff})$

lemma $\text{tl_subset_verts} : \langle \text{tl } ' (\text{arcs } G) \subseteq \text{verts } G \rangle$
using $\text{tail_eq_tl tail_in_verts}$ **by** force

lemma $\text{hd_subset_verts} : \langle \text{hd } ' (\text{arcs } G) \subseteq \text{verts } G \rangle$
using $\text{head_eq_hd head_in_verts}$ **by** force

lemma $\text{inj_on_tl} : \langle \text{inj_on uid } (\text{tl } ' (\text{arcs } G)) \rangle$
using $\text{id_vert_inj tl_subset_verts}$
by $(\text{meson inj_on_subset})$

end

definition $\text{outgoing_arcs} :: \langle ('a, 'b, 'c) \text{neural_network} \Rightarrow \text{id} \Rightarrow ('a, 'b) \text{edge set} \rangle$ **where**
 $\langle \text{outgoing_arcs } N \ n_{id} = \{a . a \in \text{arcs } (\text{graph } N) \wedge \text{uid } (\text{tl } a) = n_{id}\} \rangle$

definition $\text{neurons} :: \langle ('a, 'b, 'c) \text{neural_network} \Rightarrow ('a, 'b) \text{neuron set} \rangle$ **where**
 $\langle \text{neurons} = \text{verts } o \text{ graph} \rangle$

definition $\text{edges} :: \langle ('a, 'b, 'c) \text{neural_network} \Rightarrow ('a, 'b) \text{edge set} \rangle$ **where**
 $\langle \text{edges} = \text{arcs } o \text{ graph} \rangle$

locale $\text{nn_graph} = \text{nn_pregraph} +$
assumes $\text{id_vert_inj} : \langle \text{inj_on uid } (\text{verts } G) \rangle$

```

and inputs_In:
  <input_verts G = Set.filter (λ v. (case v of In _ ⇒ True | _ ⇒ False)) (verts G) >
and outputs_Out:
  <output_verts G = Set.filter (λ v. (case v of Out _ ⇒ True | _ ⇒ False)) (verts G) >
and internal_Neuron:
  <internal_verts G = Set.filter (λ v. (case v of Neuron _ ⇒ True | _ ⇒ False)) (verts G) >
begin

lemma nn_graph: nn_graph G by intro_locales
end

```

```

locale neural_network_digraph =
  fixes N::<('a::{comm_monoid_add,times,linorder,one}, 'b, 'c) neural_network>
  assumes <nn_graph (graph N)>
  and <φ ' {n . Neuron n ∈ (verts (graph N)) } ⊆ dom (activation_tab N)>

```

4.3.5 The empty neural network

```

definition empty::<('a, 'b) nn_pregraph> where
  <empty = (| verts={}, arcs = {}, tail = edge.tl, head = edge.hd |) >

```

```

lemma nn_pregraph_empty[simp]: <nn_pregraph (empty)>
  unfolding empty_def apply standard
  by(auto simp add: input_verts_def output_verts_def internal_verts_def)

```

```

lemma nn_graph_empty[simp]: <nn_graph (empty)>
  unfolding empty_def apply standard
  by(auto simp add: input_verts_def output_verts_def internal_verts_def)

```

```

lemma fold_inv: P e ⇒ (∀ e' x. P e' ⇒ P (f x e')) ⇒ P (fold f xs e)
  by (simp add: fold_invariant)

```

```

lemma nn_pregraph_fold: <nn_pregraph G ⇒ nn_pregraph (foldr (λ a b. add_nn_edge b a) edge_list G)>
proof(induction edge_list)
  case Nil
  then show ?case by (simp add:nn_graph.axioms)
next
  case (Cons a edge_list) note * = this
  then show ?case
    using fold_inv[of nn_pregraph G (λx s. add_nn_edge s x) edge_list]
    by (simp add: fold_inv nn_pregraph.nn_pregraph_add_nn_edge)
qed

```

```

definition
<mk_nn_pregraph edge_list = foldr (λ a b. add_nn_edge b a) edge_list empty>

```

```

lemma nn_pregraph_mk: <nn_pregraph(mk_nn_pregraph edge_list)>
  using mk_nn_pregraph_def nn_pregraph_fold
  by (metis nn_graph.axioms(1) nn_graph_empty)

```

```

lemma verts_subseteq_add_edge: nn_pregraph G ⇒ verts G ⊆ verts (add_nn_edge G a)
  unfolding add_nn_edge_def pre_digraph.add_arc_def

```

by auto

4.3.6 Computing Predictions of Neural Networks

datatype error = OK | ERROR

locale neural_network_digraph_single = neural_network_digraph N
for N::⟨('a::{comm_monoid_add,times,linorder,one}, 'b, 'a ⇒ 'a) neural_network⟩

function (sequential, domintros) predict_{digraph_single} 'n::nat
⇒ ('a::{comm_monoid_add,times,linorder,one}, 'b, 'a ⇒ 'a) neural_network
⇒ (id → 'a) ⇒ ('a, 'b) edge ⇒ ('a × error)⟩

where

⟨predict_{digraph_single} 'n inputs ((ω=_, tl=_, hd=In _)) = (o, ERROR)⟩
| ⟨predict_{digraph_single} 'n inputs ((ω=_, tl=Out_, hd=_)) = (o, ERROR)⟩
| ⟨predict_{digraph_single} 'n inputs ((ω=ω', tl=In uid_{in}, hd=_)) = (case inputs uid_{in} of
None ⇒ (o, ERROR)
| Some v ⇒ (v * ω', OK))⟩
| ⟨predict_{digraph_single} 'n N inputs e = (if o < n then
(let
ω' = ω e;
tl' = (case (tl e) of (Neuron t) ⇒ t);
E' = incoming_arcs N (Neuron.uid tl');
lvals = ((λ e'. (case predict_{digraph_single} ' (n-1) N inputs e' of
(_, ERROR) ⇒ ((o,o), ERROR)
| (v, OK) ⇒ ((v,uid (tl e')), OK))) ' E')
in
(case (activation_tab N) (φ tl') of
Some f ⇒ (ω' * (f((∑ v ∈ lvals. fst (fst v))) + (β tl')), OK)
| None ⇒ (o, ERROR)))
else (o, ERROR))⟩
apply pat_completeness
by(simp_all)

termination

by(size_change)

definition

⟨predict_{digraph_single} N inputs e = (case predict_{digraph_single} ' (card (edges N)) N inputs e of
(r, OK) ⇒ Some r
| (_, ERROR) ⇒ None)⟩

definition

⟨get_input_neuron_ids_l N = sorted_list_of_set (uid ' (input_verts (graph N)))⟩

definition

⟨mk_input_map N vs = map_of (rev (zip (get_input_neuron_ids_l N) vs))⟩

definition

⟨get_output_edge_ids_l N = sorted_list_of_set (uid ' (output_verts (graph N)))⟩

definition

⟨get_output_edge_l N = map the_elem (map (λ i. {e . e ∈ edges N ∧ i = uid (hd e)}) (get_output_edge_ids_l N))⟩

definition

⟨ predict_{digraph_single_list} N inputs' = those (map (λ e. predict_{digraph_single} N (mk_input_map N inputs') e)
(get_output_edge_l N))⟩

```

context neural_network_digraph_single begin

lemma ids_growing':  $\langle \text{neural\_network\_digraph } N \implies e \in \text{edges } N \implies \text{uid } (\text{tl } e) < \text{uid } (\text{hd } e) \rangle$ 
  by (metis comp_def edges_def neural_network_digraph_def nn_graph.axioms(1) nn_pregraph.ids_growing)

end

context neural_network_digraph begin
fun (sequential) predictdigraph:: $\langle (id \rightarrow 'a) \text{ list} \Rightarrow ('a, 'b) \text{ edge list} \Rightarrow ('a \text{ list} \times \text{error}) \rangle$ 
  where
   $\langle \text{predict}_{\text{digraph}} \_ \_ = ([], \text{ERROR}) \rangle$ 
end

record 'a data =
  inputs:: $\langle id \rightarrow 'a \rangle$ 
  outputs:: $\langle id \rightarrow 'a \rangle$ 

end

```

4.4 Main Theory (Digraph) (📄 NN_Digraph_Main)

```

theory
  NN_Digraph_Main
imports
  NN_Common
  NN_Digraph
  Activation_Functions
  Properties
begin

ML $\langle$ 
signature CONVERT_TENSORFLOW_JSON = sig
datatype neuron = In of int | Out of int
  | Neuron of {phi:TensorFlow_Type.activationT, bias:IEEEReal.decimal_approx, uid:int}
type edge = {
  tl:neuron,
  weight:IEEEReal.decimal_approx,
  hd:neuron
}
type neural_network = {
  edges: edge list,
  neurons: neuron list,
  activation_tab: TensorFlow_Type.activationT list
}
val uid_of: neuron  $\rightarrow$  int

val mk_neural_network: IEEEReal.decimal_approx TensorFlow_Type.layer list  $\rightarrow$  neural_network
end
 $\rangle$ 
ML_file $\langle$ Tools/Convert_TensorFlow_Json.ML $\rangle$ 

```

ML<

```
signature TENSORFLOW_DIGRAPH_TERM = sig
  val term_of_neuron: bool -> Activation_Term.mode -> Convert_TensorFlow_Json.neuron -> term
end
>
```

ML_file <Tools/TensorFlow_Digraph_Term.ML>

ML<

```
signature CONVERT_TENSORFLOW_DIGRAPH =
  sig
    val def_nn: Activation_Term.mode -> string -> 'a -> 'b -> Nano_Json_Type.json -> local_theory ->
Proof.context
  end
>
```

ML_file<Tools/Convert_TensorFlow_Digraph.ML>

ML<

```
val _ = Theory.setup
  (Convert_TensorFlow_Symtab.add_encoding(digraph,
    Convert_TensorFlow_Digraph.def_nn Activation_Term.MultiList))
val _ = Theory.setup
  (Convert_TensorFlow_Symtab.add_encoding(digraph_single,
    Convert_TensorFlow_Digraph.def_nn Activation_Term.Single))
>
```

end

5 Neural Networks as Layers

5.1 Preliminaries

5.1.1 Useful Definitions for Analysing Matrix Predictions (Prediction_Utils_Matrix)

theory

Prediction_Utils_Matrix

imports

Complex_Main

Jordan_Normal_Form.Matrix

begin

definition $max_{mat} :: \langle 'a :: linorder Matrix.mat \Rightarrow 'a \rangle$ where
 $\langle max_{mat} = Max\ o\ elements_mat \rangle$

definition $min_{mat} :: \langle 'a :: linorder Matrix.mat \Rightarrow 'a \rangle$ where
 $\langle min_{mat} = Min\ o\ elements_mat \rangle$

lemma $finite_elements_mat$: $finite\ (elements_mat\ A)$
unfolding $elements_mat_def$ **by** blast

lemma $max_{mat_is_element}$:
shows $\langle elements_mat\ m \neq \{\} \implies max_{mat}\ m \in elements_mat\ m \rangle$
by (simp add: finite_elements_mat max_mat_def)

lemma $min_{mat_is_element}$:
 $\langle elements_mat\ m \neq \{\} \implies min_{mat}\ m \in elements_mat\ m \rangle$
by (simp add: finite_elements_mat min_mat_def)

definition $max_list :: 'a :: linorder\ list \Rightarrow 'a$ where
 $max_list\ xs = fold\ max\ xs\ (hd\ xs)$

definition $min_list :: 'a :: linorder\ list \Rightarrow 'a$ where
 $min_list\ xs = fold\ min\ xs\ (hd\ xs)$

definition $max_{vec} :: \langle 'a :: linorder Matrix.vec \Rightarrow 'a \rangle$ where
 $\langle max_{vec} = max_list\ o\ list_of_vec \rangle$

definition $min_{vec} :: \langle 'a :: linorder Matrix.vec \Rightarrow 'a \rangle$ where
 $\langle min_{vec} = min_list\ o\ list_of_vec \rangle$

lemma $max_{vec_is_element}$:
shows $\langle list_of_vec\ m \neq [] \implies max_{vec}\ m \in set\ (list_of_vec\ m) \rangle$
apply (simp add: max_vec_def max_list_def)
by (metis List.finite_set Max.set_eq_fold Max_eq_iff hd_in_set list.discl set_ConsD set_empty2)

lemma $\text{min}_{vec_is_element}$:
shows $\langle \text{list_of_vec } m \neq [] \implies \text{min}_{vec} m \in \text{set}(\text{list_of_vec } m) \rangle$
apply (simp add: min_{vec_def} min_list_def)
by (metis List.finite_set Min.set_eq_fold Min.in_empty_iff insert_absorb $\text{list.set_sel}(1)$ $\text{list.simps}(15)$)

lemma $\text{max}_{vec_vCons_append_eq}$: $\langle \text{max}_{vec} (\text{vCons } x \text{ } xs) = \text{max}_{vec} xs \vee \text{max}_{vec} (\text{vCons } x \text{ } xs) = x \rangle$
proof(cases $\text{max}_{vec} (\text{vCons } x \text{ } xs) = x$)
case True
then show ?thesis **by** blast
next
case False
then show ?thesis **apply**(simp add: max_{vec_def} max_list_def Max.eq_iff insertCI insertE)
by (smt (verit) List.finite_set Max.in_idem Max.set_eq_fold Max.insert comp_apply
 $\text{fold_simps}(1)$ $\text{list.simps}(15)$ $\text{list.simps}(3)$ list_of_vec_vCons max.right_idem max_{vec_def}
 $\text{max}_{vec_is_element}$ max_list_def set_ConsD set_empty2)

qed

lemma $\text{max}_{vec_append_eq}$: $\langle \text{max}_{vec} (\text{vec_of_list } (xs @ [x])) = \text{max}_{vec} (\text{vec_of_list } xs) \vee \text{max}_{vec} (\text{vec_of_list } (xs @ [x])) = x \rangle$
proof(cases $\text{max}_{vec} (\text{vec_of_list } (xs @ [x])) = x$)
case True
then show ?thesis **by** blast
next
case False
then show ?thesis
apply(simp add: max_{vec_def} $\text{max}_{vec_is_element}$ max_list_def)
by (metis (no_types, lifting) append_self_conv2 comp_apply $\text{fold.simps}(1)$
 fold_append $\text{fold_simps}(2)$ hd_append2 id_apply $\text{list.sel}(1)$ list_vec max_def)

qed

lemma $\text{min}_{vec_vCons_append_eq}$: $\langle \text{min}_{vec} (\text{vCons } x \text{ } xs) = \text{min}_{vec} xs \vee \text{min}_{vec} (\text{vCons } x \text{ } xs) = x \rangle$
proof(cases $\text{min}_{vec} (\text{vCons } x \text{ } xs) = x$)
case True
then show ?thesis **by** blast
next
case False
then show ?thesis
apply(simp add: min_{vec_def} min_list_def Min.eq_iff insertCI insertE)
by (metis (no_types, lifting) List.finite_set Min.in_idem Min.insert
 $\text{Min.semilattice_set_axioms}$ Min_def $\text{fold_simps}(1)$ $\text{list.set_sel}(1)$ $\text{list.simps}(15)$
 min_def $\text{semilattice_set.set_eq_fold}$ set_empty)

qed

lemma $\text{min}_{vec_append_eq}$: $\langle \text{min}_{vec} (\text{vec_of_list } (xs @ [x])) = \text{min}_{vec} (\text{vec_of_list } xs) \vee \text{min}_{vec} (\text{vec_of_list } (xs @ [x])) = x \rangle$
proof(cases $\text{min}_{vec} (\text{vec_of_list } (xs @ [x])) = x$)
case True
then show ?thesis **by** blast
next
case False
then show ?thesis

```

apply(simp add: minvec_def minvec_is_element min_list_def)
by (metis (no_types, lifting) append_self_conv2 comp_apply fold_simps(1)
  fold_append fold_simps(2) hd_append2 id_apply list.sel(1) list_vec min_def)
qed

```

```

lemma maxvec_vec_cons_eq: <maxvec ((vec_of_list [x]) @v xs) = maxvec xs ∨ maxvec ((vec_of_list [x]) @v xs) = x>
proof(cases maxvec ((vec_of_list [x]) @v xs) = x)
  case True
  then show ?thesis by blast
next
  case False
  then show ?thesis
  apply(simp add: maxvec_def max_list_def)
  using max_def
  by (metis (no_types, lifting) comp_apply fold_simps(2) list.sel(1)
    list_of_vec_vCons maxvec_def maxvec_vCons_append_eq max_list_def)
qed

```

```

lemma maxvec_cons_eq: <maxvec (vec_of_list (x#xs)) = maxvec (vec_of_list xs) ∨ maxvec (vec_of_list (x#xs)) = x>
proof(cases maxvec (vec_of_list (x#xs)) = x)
  case True
  then show ?thesis by blast
next
  case False
  then show ?thesis
  apply(simp add: maxvec_def maxvec_is_element max_list_def)
  by (metis (no_types, lifting) comp_apply fold_simps(2) list.sel(1)
    list_of_vec_vCons max.idem maxvec_def maxvec_vCons_append_eq max_list_def)
qed

```

```

lemma minvec_vec_cons_eq: <minvec ((vec_of_list [x]) @v xs) = minvec xs ∨ minvec ((vec_of_list [x]) @v xs) = x>
proof(cases minvec ((vec_of_list [x]) @v xs) = x)
  case True
  then show ?thesis by blast
next
  case False
  then show ?thesis
  apply(simp add: maxvec_def)
  using minvec_vCons_append_eq by blast
qed

```

```

lemma minist_cons_eq: <minvec (vec_of_list (x#xs)) = minvec (vec_of_list xs) ∨ minvec (vec_of_list (x#xs)) = x>
proof(cases minvec (vec_of_list (x#xs)) = x)
  case True
  then show ?thesis by blast
next
  case False
  then show ?thesis
  apply (simp add: minvec_def minvec_is_element min_list_def)
  by (metis (no_types, lifting) comp_apply fold_simps(2) list.sel(1)
    list_of_vec_vCons min.idem minvec_def minvec_vCons_append_eq min_list_def)
qed

```

```

lemma maxvec_vec_append_limit: assumes <xs ≠ vNil> shows <maxvec xs ≤ maxvec (vCons x xs)>

```

```

proof(cases  $\max_{vec} (vCons\ x\ xs) = x$ )
  case True
  then show ?thesis using assms
  apply (simp add:  $\max_{vec\_def}$   $\max\_list\_def$ )
  by (metis List.finite_set Max.set_eq_fold Max.insert hd_in_set insert_absorb
    list.simps(15)  $\max.order1$  set_empty2 vec_list vec_of_list_Nil)
next
  case False
  then show ?thesis
  using dual_order.order_iff_strict  $\max_{vec\_vCons\_append\_eq}$  by auto
qed

lemma  $\max_{vec\_append\_limit}$ : assumes  $\langle xs \neq [] \rangle$  shows  $\langle \max_{vec} (vec\_of\_list\ xs) \leq \max_{vec} (vec\_of\_list\ (xs\ @\ [x])) \rangle$ 
proof(cases  $\max_{vec} (vec\_of\_list\ (xs\ @\ [x])) = x$ )
  case True
  then show ?thesis using assms
  apply (simp add:  $\max_{vec\_def}$   $\max_{vec\_is\_element}$   $\max\_list\_def$  )
  by (metis List.finite_set Max.set_eq_fold Max.ge comp_apply list.set_intros(2)
    list_vec  $\max_{vec\_def}$   $\max_{vec\_is\_element}$   $\max\_list\_def$  rotate1.simps(2) set_rotate1)
next
  case False
  then show ?thesis
  by (metis dual_order.refl  $\max_{vec\_append\_eq}$ )
qed

lemma  $\min_{vec\_vec\_append\_limit}$ : assumes  $\langle xs \neq vNil \rangle$  shows  $\langle \min_{vec}\ xs \geq \min_{vec} (vCons\ x\ xs) \rangle$ 
proof(cases  $\min_{vec} (vCons\ x\ xs) = x$ )
  case True
  then show ?thesis using assms
  apply (simp add:  $\min_{vec\_def}$   $\min\_list\_def$ )
  by (metis List.finite_set Min.insert Min.set_eq_fold
    Orderings.order_eq_iff hd_in_set insert_absorb list.simps(15)
    min_def set_empty2 vec_list vec_of_list_Nil)
next
  case False
  then show ?thesis
  by (metis dual_order.refl  $\min_{vec\_vCons\_append\_eq}$ )
qed

lemma  $\min_{vec\_append\_limit}$ : assumes  $\langle xs \neq [] \rangle$  shows  $\langle \min_{vec} (vec\_of\_list\ xs) \geq \min_{vec} (vec\_of\_list\ (xs\ @\ [x])) \rangle$ 
proof(cases  $\min_{vec} (vec\_of\_list\ (xs\ @\ [x])) = x$ )
  case True
  then show ?thesis using assms
  apply (simp add:  $\min_{vec\_def}$   $\min_{vec\_is\_element}$   $\min\_list\_def$  )
  by (metis (no_types, lifting) append.right_neutral comp_apply
    fold_append fold_simps(2) hd_append2 linorder_linear list_vec min_def)
next
  case False
  then show ?thesis
  by (metis dual_order.refl  $\min_{vec\_append\_eq}$ )
qed

lemma  $\max_{vec\_vec\_cons\_limit}$ : assumes  $\langle xs \neq vNil \rangle$  shows  $\langle \max_{vec}\ xs \leq \max_{vec} ((vec\_of\_list\ [x])\ @_v\ xs) \rangle$ 
proof(cases  $\max_{vec} ((vec\_of\_list\ [x])\ @_v\ xs) = x$ )

```

```

case True
then show ?thesis using assms
  by (metis append_vec_vCons append_vec_vNil max_vec_vec_append_limit vec_of_list_Cons
    vec_of_list_Nil)
next
case False
then show ?thesis
  by (simp add: assms max_vec_vec_append_limit)
qed

lemma max_vec_cons_limit: assumes <xs ≠ []> shows <max_vec (vec_of_list xs) ≤ max_vec (vec_of_list (x#xs))>
proof(cases max_vec (vec_of_list (x#xs)) = x)
case True
then show ?thesis using assms
  by (metis list_vec max_vec_vec_append_limit vec_of_list_Cons vec_of_list_Nil)
next
case False
then show ?thesis
  by (metis dual_order_refl max_vec_cons_eq)
qed

lemma min_vec_vec_cons_limit: assumes <xs ≠ vNil> shows <min_vec xs ≥ min_vec ((vec_of_list [x]) @_v xs)>
proof(cases min_vec ((vec_of_list [x]) @_v xs) = x)
case True
then show ?thesis using assms
  by (metis append_vec_vCons append_vec_vNil min_vec_vec_append_limit vec_of_list_Cons
    vec_of_list_Nil)
next
case False
then show ?thesis
  by (simp add: assms min_vec_vec_append_limit)
qed

lemma min_vec_cons_limit: assumes <xs ≠ []> shows <min_vec (vec_of_list xs) ≥ min_vec (vec_of_list (x#xs))>
proof(cases min_vec (vec_of_list (x#xs)) = x)
case True
then show ?thesis using assms
  by (metis list_vec min_vec_vec_append_limit vec_of_list_Cons vec_of_list_Nil)
next
case False
then show ?thesis
  by (metis min_list_cons_eq order_refl)
qed

```

Converting Predictions to Percentages **definition** `prediction2percentage :: <real Matrix.vec ⇒ real Matrix.vec> where`
`<prediction2percentage m = (let m' = max_vec m in map_vec (λ e. e / m' * 100.0) m)>`

```

lemma prediction2percentage_id:
  assumes <max_vec p = 100>
  shows <prediction2percentage p = p>
  using assms unfolding prediction2percentage_def
  by auto

```

Maximum Prediction **definition** $\text{posmax_of} :: \langle 'a::\text{linorder Matrix.vec} \Rightarrow (\text{nat} \times 'a) \text{ option} \rangle$ **where**
 $\langle \text{posmax_of } l = (\text{let } m = \max_{\text{vec}} l \text{ in find } (\lambda e. \text{snd } e = m) (\text{enumerate } o (\text{list_of_vec } l))) \rangle$
definition $\text{pos_of_max} :: \langle 'a::\text{linorder Matrix.vec} \Rightarrow \text{nat option} \rangle$ **where**
 $\langle \text{pos_of_max } l = \text{map_option fst } (\text{posmax_of } l) \rangle$

definition $\text{posmax_of}' :: \langle 'a::\text{linorder Matrix.vec} \Rightarrow (\text{nat} \times 'a) \text{ option} \rangle$ **where**
 $\langle \text{posmax_of}' l = (\text{let } l' = \text{list_of_vec } l \text{ in}$
 $\quad (\text{if } l' = [] \text{ then None}$
 $\quad \quad \text{else Some } ((\text{hd } o \text{ rev } o (\text{sort_key snd}) o (\text{enumerate } o)) l')) \rangle$

definition $\text{pos_of_max}' :: \langle 'a::\text{linorder Matrix.vec} \Rightarrow \text{nat option} \rangle$ **where**
 $\langle \text{pos_of_max}' l = \text{map_option fst } (\text{posmax_of}' l) \rangle$

lemma $\text{find_append_eq} :: \langle \text{find } P (xs@[x]) = (\text{if find } P xs = \text{None} \text{ then find } P [x] \text{ else find } P xs) \rangle$

proof (induction xs)

case Nil

then show ?case by simp

next

case (Cons a xs)

then show ?case by simp

qed

Distance of Maximum Prediction to Next Highest Prediction **definition** $\delta_{\text{min}} :: \text{real mat} \Rightarrow \text{real}$ **where**
 $\langle \delta_{\text{min}} m = (\text{let } m' = \max_{\text{mat}} m; m'' = \text{Max } (\text{elements_mat } m - \{m'\})$
 $\quad \text{in } |m' - m''|) \rangle$

end

5.1.2 Desirable Properties of Neural Networks Predictions (⊖ Properties_Matrix)

theory Properties_Matrix

imports

Properties

Prediction_Utils_Matrix

Jordan_Normal_Form.Matrix

begin

definition $\text{zip_vec} :: 'a \text{ Matrix.vec} \Rightarrow 'b \text{ Matrix.vec} \Rightarrow ('a \times 'b) \text{ Matrix.vec}$ **where**
 $\text{zip_vec } A B \equiv \text{Matrix.vec } (\text{dim_vec } A) (\lambda i. ((A \$ i), (B \$ i)))$

fun $\text{map_vec2} :: \langle ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \text{ Matrix.vec} \Rightarrow 'b \text{ Matrix.vec} \Rightarrow 'c \text{ Matrix.vec} \rangle$
where
 $\text{map_vec2 } f xs ys = \text{map_vec } (\lambda (x,y). f x y) (\text{zip_vec } xs ys)$

fun $\text{checkget_result_mat}$ **where**

$\langle \text{checkget_result_mat } _ \text{ None } _ = (\text{None}, \text{True}) \rangle$

$\langle \text{checkget_result_mat } \varepsilon (\text{Some } xs) (\text{Some } ys) = (\text{Some } xs, \text{fold } (\wedge) (\text{list_of_vec } (\text{map_vec2 } (\lambda x y. x \approx[\varepsilon] \approx y) xs ys))$
 $\text{True}) \rangle$

$\langle \text{checkget_result_mat } _ r = (r, \text{False}) \rangle$

definition $\langle \text{check_result_mat } r \varepsilon s = \text{snd } (\text{checkget_result_mat } \varepsilon r s) \rangle$

notation $\text{check_result_mat } (((_)/ \approx[_]) \approx_m (_)) [60, 60] 60$

definition $\text{ensure_testdata_range_mat} :: \langle \text{real} \Rightarrow \text{real Matrix.vec list} \Rightarrow (\text{real Matrix.vec} \rightarrow \text{real Matrix.vec}) \Rightarrow \text{real Matrix.vec list} \Rightarrow \text{bool} \rangle$

where

```
⟨ensure_testdata_range_mat delta inputs P outputs
  = foldl (∧) True
    (map (λ e. (P (fst e)) ≈[delta]≈m Some (snd e))
      (zip inputs outputs))⟩
notation ensure_testdata_range_mat ((_) ⊨m {( _ ) } ( _ ) {( _ ) } [61, 3, 90, 3] 60)
```

Interval Arithmetic **definition** ⟨intervals_of_mat δ A = Matrix.vec (dim_vec A) (λ i. Interval((A*i*)−|δ|,(A*i*)+|δ|)) ⟩

definition ⟨intervals_of_m δ = map (intervals_of_mat δ) ⟩

```
fun check_result_mat_interval_mat :: ⟨'a::preorder Matrix.vec option ⇒ 'a interval Matrix.vec option ⇒ bool⟩ where
  ⟨check_result_mat_interval_mat None None      = True⟩
  | ⟨check_result_mat_interval_mat (Some xs) (Some ys) = fold (∧) (list_of_vec (map_vec2 (λ x y. x ∈ set_of y) xs ys))
    True⟩
  | ⟨check_result_mat_interval_mat _ _ = False⟩
```

notation check_result_mat_interval_mat (((_) / ≈_m (_)) [60, 60] 60)

We define *check_result_mat_interval* for checking that two matrices are approximatively equal (we need the error interval due to possible rounding errors in IEEE754 arithmetic in python compared to mathematical reals in Isabelle).

definition *ensure_testdata_interval_mat* :: ⟨real Matrix.vec list ⇒ (real Matrix.vec → real Matrix.vec) ⇒ real interval Matrix.vec list ⇒ bool⟩

where

```
⟨ensure_testdata_interval_mat inputs P outputs
  = foldl (∧) True
    (map (λ e. let a = (P (fst e)) in let b = Some (snd e) in (a ≈m b))
      (zip inputs outputs))⟩
notation ensure_testdata_interval_mat (⊨im {( _ ) } ( _ ) {( _ ) } [3, 90, 3] 60)
```

Using *check_result_mat_interval* we now define the property *ensure_testdata_interval* to check that the (symbolically) computed predictions of a neural network meet our expectations.

Maximum Classifiers

definition

```
ensure_testdata_max_mat :: ⟨real Matrix.vec list ⇒ (real Matrix.vec → real Matrix.vec) ⇒ real Matrix.vec list ⇒ bool⟩
where
⟨ensure_testdata_max_mat inputs P outputs
  = foldl (∧) True
    (map (λ e. case P (fst e) of
      None ⇒ False
      | Some p ⇒ pos_of_max p = pos_of_max (snd e))
      (zip inputs outputs))⟩
notation ensure_testdata_max_mat (⊨m {( _ ) } ( _ ) {( _ ) } [3, 90, 3] 60)
```

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, this can be tested using *ensure_testdata_max*

end

5.1.3 Sequential Layers (NN_Layers)

theory

NN_Layers

imports

Activation_Functions

begin

In this theory, we model feed-forward neural networks as “computational layers” following the structure of TensorFlow [1] closely.

```
record InOutRecord =  
  name:: String.literal  
  units:: nat
```

```
record ('b) ActivationRecord = InOutRecord +  
   $\varphi$  :: 'b
```

```
record ('a, 'b, 'c) LayerRecord = <'b> ActivationRecord +  
   $\beta$  :: <'a>  
   $\omega$  :: <'c>
```

```
datatype ('a, 'b, 'c) layer = In <InOutRecord>  
  | Out <InOutRecord>  
  | Dense <('a, 'b, 'c) LayerRecord>  
  | Activation <('b) ActivationRecord>
```

fun *units_l* **where**

```
<unitsl (In l) = units l  
| <unitsl (Out l) = units l  
| <unitsl (Dense l) = units l  
| <unitsl (Activation l) = units l
```

lemmas [*nn_layer*] = *InOutRecord.simps* *ActivationRecord.simps* *LayerRecord.simps* *layer.simps* *units_l.simps*

The datatype *layer* models the currently supported layer types

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 *In input* (input layer), *Out output* (output layer), *Dense dense_layer* (dense layer), and *Activation activation_layer* (activation layer). As we allow for the abstraction of activation functions, we abstract from the actual type for the activation function (modelled by the type variable 'b and from the actual type of weight and bias (modelled by the type variables 'a and 'c respectively).

Therefore, we do not need to model TensorFlow’s Lambda layer explicitly (which is TensorFlow’s mechanism for supporting custom activation functions).

Each *LayerRecord* contains the activation, weights and bias in our network φ , β and ω respectively), while our *ActivationRecord* only contains our abstracted activation function.

fun *isIn* **where**

```
<isIn (In _) = True>  
| <isIn _ = False>
```

fun *isOut* **where**

```

  <isOut (Out _) = True>
| <isOut _ = False>

```

fun isInternal where

```

  <isInternal (Out _) = False>
| <isInternal (In _) = False>
| <isInternal _ = True>

```

lemma isInternal': $\langle \text{isInternal } n = (\neg (\text{isIn } n) \wedge \neg (\text{isOut } n)) \rangle$

```

by (metis isIn.elims(2) isIn.simps(1) isInternal.elims(3) isInternal.simps(1) isInternal.simps(2)
      isOut.elims(2) isOut.simps(1))

```

record ('a, 'b, 'c) *neural_network_seq_layers* =

```

  layers :: <('a, 'b, 'c) layer list>
  activation_tab :: <'b  $\Rightarrow$  (('a  $\Rightarrow$  'a) option)>

```

lemmas [nn_layer] = *neural_network_seq_layers.simps*

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

fun out_deg_layer

where

```

  <out_deg_layer (In l) = (units l)>
| <out_deg_layer (Out l) = (units l)>
| <out_deg_layer (Activation l) = units b>
| <out_deg_layer (Dense l) = units b>

```

fun units_layer where

```

  <units_layer (In l) = units b>
| <units_layer (Out l) = units b>
| <units_layer (Activation l) = units b>
| <units_layer (Dense l) = units b>

```

fun φ _layer where

```

  < $\varphi$ _layer (In l) = None>
| < $\varphi$ _layer (Out l) = None>
| < $\varphi$ _layer (Activation l) = Some ( $\varphi$  l)>
| < $\varphi$ _layer (Dense l) = Some ( $\varphi$  l)>

```

fun in_deg_layer where

```

  in_deg_layer (In l) = units l
| in_deg_layer (Out l) = units l
| in_deg_layer (Activation l) = units l
| in_deg_layer (Dense l) = length ( $\omega$  l ! o)

```

lemmas [nn_layer] = *out_deg_layer.simps units_layer.simps φ _layer.simps*

definition

```

  <out_deg_NN N = (if layers N = [] then o else (units_layer  $\circ$  last  $\circ$  layers) N)>

```

definition

```
<in_deg_NN N = (if layers N = [] then 0 else (units_layer ∘ hd ∘ layers) N)>
```

ML<

```
signature CONVERT_TENSORFLOW_SEQ_LAYER =
```

```
sig
```

```
  type layer_config = {ActivationRecordC: term, DenseC: term, InC: term, InOutRecordC: term, LayerRecordC: term, OutC: term, activation_term: Activation_Term.mode, biasT_conv: term -> term, layer_def: thm, layer_extC: term, layersT: typ, locale_name: string, ltype: typ, phiT: typ, valid_activation_tab: thm, weightsT_conv: term -> int -> term}
  val def_seq_layer_nn: layer_config -> bstring -> typ -> typ -> Nano_Type.json -> local_theory
    -> Proof.context
```

```
end
```

```
>
```

```
ML_file<Tools/Convert_TensorFlow_Seq_Layer.ML>
```

```
end
```

5.1.4 Neural Network Lipschitz Continuity

theory

```
  NN_Lipschitz_Continuous
```

imports

```
  NN_Layers
```

```
  HOL—Library.Numeral_Type
```

```
  Activation_Functions
```

```
  Matrix_Utils
```

```
  HOL—Analysis.Analysis
```

```
begin
```

Lipschitz Continuity of Functions (real)

Splitting Function

```
Neural Network: Activations lemma relu_lipschitz: 1—lipschitz_on (X::real set) (relu)
```

```
  unfolding lipschitz_on_def relu_def dist_real_def
```

```
  by auto
```

```
lemma identity_lipschitz: 1—lipschitz_on (X::real set) (identity)
```

```
  unfolding lipschitz_on_def identity_def dist_real_def
```

```
  by auto
```

```
Neural Network: Layers lemma input_output_lipschitz_continuous:
```

```
<1—lipschitz_on (U::real set) (λ i. i)>
```

```
using lipschitz_intros by simp
```

```
lemma activation_lipschitz_continuous:
```

```
  assumes <C—lipschitz_on U f>
```

```
  shows <C—lipschitz_on U (λ i. f i)>
```

```
  using assms by simp
```

```
lemma lipschitz_on_add_const:
```

shows $(1::\text{real})\text{--lipschitz_on } (U::\text{real set}) (\lambda x. x + c)$
unfolding `lipschitz_on_def` **by** `simp`

lemma `lipschitz_on_fold_add`:
shows $1\text{--lipschitz_on } (U::\text{real set}) (\text{fold } (+) \text{ xs})$
unfolding `lipschitz_on_def`
by `(simp add: fold_plus_sum_list_rev)`

lemma `lipschitz_on_fold_add_zero`:
shows $1\text{--lipschitz_on } (U::\text{real set}) (\lambda x. \text{fold } (+) [x] (o::\text{real}))$
unfolding `lipschitz_on_def`
by `(simp add: fold_plus_sum_list_rev foldr_conv_fold)`

lemma `lipschitz_on_foldr_add`:
shows $1\text{--lipschitz_on } (U::\text{real set}) (\lambda s. \text{foldr } (+) \text{ xs } s)$
unfolding `lipschitz_on_def`
by `(simp add: fold_plus_sum_list_rev foldr_conv_fold)`

lemma `lipschitz_on_sumlist_rev`:
shows $1\text{--lipschitz_on } (U::\text{real set}) ((+) (\text{sum_list } (\text{rev } \text{xs})))$
using `fold_plus_sum_list_rev lipschitz_on_fold_add` **by** `metis`

lemma `lipschitz_on_sumlist`:
shows $1\text{--lipschitz_on } (U::\text{real set}) ((+) (\text{sum_list } \text{xs}))$
using `fold_plus_sum_list_rev lipschitz_on_fold_add`
by `(metis sum_list_rev)`

lemma `lipschitz_on_mult_const`:
shows $|c|\text{--lipschitz_on } (U::\text{real set}) (\lambda x. x * c)$
unfolding `lipschitz_on_def dist_real_def` **using** `abs_mult`
by `(metis (no_types, opaque_lifting) Orderings.order_eq_iff_abs_ge_zero mult.commute real_scaleR_def scaleR_right_diff_distrib)`

lemma `lipschitz_on_weighted_sum_single`:
 $|w|\text{--lipschitz_on } (U::\text{real set}) (\lambda x. x * w + b)$
unfolding `lipschitz_on_def dist_real_def` **using** `abs_mult`
by `(metis Groups.mult_ac(2) abs_ge_zero add_diff_cancel_right left_diff_distrib order.refl)`

lemma `lipschitz_on_fold_add_zero'`:
shows $2\text{--lipschitz_on } (U::\text{real set}) (\lambda x. (\text{fold } (+) [x,x] (o::\text{real})) + w)$
unfolding `lipschitz_on_def`
by `(metis (no_types, lifting) ext List.fold_simps(1,2) add.commute dist_add_cancel dist_triangle_add mult.commute mult_2_right zero_le_natural)`

lemma `lipschitz_on_mult_const'`:
shows $\langle \forall x \in \text{set } \text{xs}. |c|\text{--lipschitz_on } (\text{set } \text{xs}) (\lambda y. c * y) \rangle$
using `lipschitz_on_mult_const`
by `(metis mult_commute_abs)`

typedef $('a, 'nr::\text{finite}, 'nc::\text{finite}) \text{fixed_mat} =$
 $\text{carrier_mat } (\text{CARD } ('nr)) (\text{CARD } ('nc)) :: 'a \text{ mat set}$
morphisms `Rep_fixed_mat Abs_fixed_mat` **by** `blast`

setup_lifting `type_definition_fixed_mat`

```

typedef ('a, 'n::finite) fixed_vec =
  carrier_vec (CARD('n)) :: 'a vec set
morphisms Rep_fixed_vec Abs_fixed_vec
using vec_carrier by blast

```

```

setup_lifting type_definition_fixed_vec

```

```

definition dim_vecf :: ('a, 'n::finite) fixed_vec  $\Rightarrow$  nat where
  dim_vecf v = CARD('n)

```

```

definition dim_colf :: ('a, 'nc::finite, 'nr::finite) fixed_mat  $\Rightarrow$  nat where
  dim_colf m = CARD('nc)

```

```

definition dim_rowf :: ('a, 'nc::finite, 'nr::finite) fixed_mat  $\Rightarrow$  nat where
  dim_rowf m = CARD('nr)

```

```

definition fixed_mat_zero :: ('a::zero, 'nr::finite, 'nc::finite) fixed_mat where
  fixed_mat_zero = Abs_fixed_mat (om (CARD('nr)) (CARD('nc)))

```

```

definition fixed_mat_add :: ('a::plus, 'nr::finite, 'nc::finite) fixed_mat  $\Rightarrow$  ('a, 'nr, 'nc) fixed_mat  $\Rightarrow$  ('a, 'nr, 'nc)
  fixed_mat where
  fixed_mat_add A B = Abs_fixed_mat (Rep_fixed_mat A + Rep_fixed_mat B)

```

```

definition fixed_vec_zero :: ('a::zero, 'nr::finite) fixed_vec where
  fixed_vec_zero = Abs_fixed_vec (ov (CARD('nr)))

```

```

definition fixed_vec_add :: ('a::plus, 'nr::finite) fixed_vec  $\Rightarrow$  ('a, 'nr) fixed_vec  $\Rightarrow$  ('a, 'nr) fixed_vec where
  fixed_vec_add A B = Abs_fixed_vec (Rep_fixed_vec A + Rep_fixed_vec B)

```

```

lift_definition fixed_mat_smult :: 'a::times  $\Rightarrow$  ('a, 'nr::finite, 'nc::finite) fixed_mat  $\Rightarrow$  ('a, 'nr, 'nc) fixed_mat
is  $\lambda$  c A. c  $\cdot$ m A
using smult_carrier_mat .

```

```

lift_definition fixed_mat_index :: ('a, 'nr::finite, 'nc::finite) fixed_mat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  'a
is  $\lambda$  A i j. A $$$ (i, j) .

```

```

lift_definition fixed_vec_index :: ('a, 'nr::finite) fixed_vec  $\Rightarrow$  nat  $\Rightarrow$  'a
is vec_index .

```

```

lift_definition fixed_vec_smult :: 'a::times  $\Rightarrow$  ('a, 'nr::finite) fixed_vec  $\Rightarrow$  ('a, 'nr) fixed_vec
is  $\lambda$  c A. c  $\cdot$ v A
using smult_carrier_mat by simp

```

```

lift_definition mult_vec_fixed_mat ::
  ('a::semiring_o, 'nr::finite) fixed_vec  $\Rightarrow$  ('a, 'nr, 'nc::finite) fixed_mat  $\Rightarrow$  ('a, 'nc) fixed_vec
  (infixl fv* 70)
is  $\lambda$  v A. vec (dim_col A) ( $\lambda$  i. col A i  $\cdot$  v)
by simp

```

```

lift_definition map_fixed_vec :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  ('a, 'nr::finite) fixed_vec  $\Rightarrow$  ('b, 'nr::finite) fixed_vec
is map_vec :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  'a vec  $\Rightarrow$  'b vec
by simp

```

lemma *zero_in_carrier*:

$o_m (\text{CARD}('nr)) (\text{CARD}('nc)) \in \text{carrier_mat} (\text{CARD}('nr)) (\text{CARD}('nc))$
using *zero_carrier_mat* **by** *simp*

lemma *Rep_fixed_mat_zero* [*simp*]:

$\text{Rep_fixed_mat} (\text{fixed_mat_zero} :: ('a::\text{zero}, 'nr::\text{finite}, 'nc::\text{finite}) \text{fixed_mat}) = o_m (\text{CARD}('nr)) (\text{CARD}('nc))$
unfolding *fixed_mat_zero_def*
using *zero_in_carrier*
by (*simp add: Abs_fixed_mat_inverse*)

lemma *Rep_fixed_mat_add* [*simp*]:

$\text{Rep_fixed_mat} (\text{fixed_mat_add} A B) = \text{Rep_fixed_mat} A + \text{Rep_fixed_mat} B$
unfolding *fixed_mat_add_def*
apply (*subst Abs_fixed_mat_inverse*)
using *Rep_fixed_mat*
apply *fastforce* **by** *auto*

lemma *Rep_fixed_vec_zero* [*simp*]:

$\text{Rep_fixed_vec} (\text{fixed_vec_zero} :: ('a::\text{zero}, 'n::\text{finite}) \text{fixed_vec}) = o_v (\text{CARD}('n))$
unfolding *fixed_vec_zero_def*
using *zero_in_carrier*
by (*simp add: Abs_fixed_vec_inverse*)

lemma *Rep_fixed_vec_add* [*simp*]:

$\text{Rep_fixed_vec} (\text{fixed_vec_add} A B) = \text{Rep_fixed_vec} A + \text{Rep_fixed_vec} B$
unfolding *fixed_vec_add_def*
apply (*subst Abs_fixed_vec_inverse*)
subgoal **using** *Rep_fixed_vec*
using *add_carrier_vec* **by** *blast*
subgoal **by** *blast*
done

lemma *Rep_fixed_mat_inject*: $\text{Rep_fixed_mat} A = \text{Rep_fixed_mat} B \implies A = B$

by (*simp add: Rep_fixed_mat_inject*)

lemma *Rep_fixed_vec_inject*: $\text{Rep_fixed_vec} A = \text{Rep_fixed_vec} B \implies A = B$

by (*simp add: Rep_fixed_vec_inject*)

lift_definition *row_fixed* :: $('a, 'n::\text{finite}, 'm::\text{finite}) \text{fixed_mat} \Rightarrow \text{nat} \Rightarrow ('a, 'm) \text{fixed_vec}$ **is**

$\lambda A i. \text{vec} (\text{CARD}('m)) (\lambda j. A \$\$ (i, j))$

by *simp*

lift_definition *col_fixed* :: $('a, 'n::\text{finite}, 'm::\text{finite}) \text{fixed_mat} \Rightarrow \text{nat} \Rightarrow ('a, 'n) \text{fixed_vec}$ **is**

$\lambda A j. \text{vec} (\text{CARD}('n)) (\lambda i. A \$\$ (i, j))$

by *simp*

lemma $\text{CARD}(285) = 285$ **by** *auto*

instantiation *fixed_mat* :: $(\text{semiring}_1, \text{finite}, \text{finite}) \text{times}$

begin

lift_definition *mat_mult* :: $('a::\text{semiring}_1, 'n::\text{finite}, 'm::\text{finite}) \text{fixed_mat} \Rightarrow$

$('a, 'm, 'k::\text{finite}) \text{fixed_mat} \Rightarrow$

$('a, 'n, 'k) \text{fixed_mat}$ **is**

```

λA B. mat (CARD('n)) (CARD('k)) (λ(i,j).
  sum_list (map (λl. A $$ (i,l) * B $$ (l,j)) [o..<CARD('m)]))
using mat_carrier by blast

```

instance ..

end

instantiation fixed_mat :: ({real_normed_vector, times, one, real_algebra_1}, finite, finite) real_normed_vector
begin

definition zero_fixed_mat :: ('a, 'nr::finite, 'nc::finite) fixed_mat **where**
 zero_fixed_mat = fixed_mat_zero

definition plus_fixed_mat :: ('a, 'nr::finite, 'nc::finite) fixed_mat ⇒ ('a, 'nr, 'nc) fixed_mat ⇒ ('a, 'nr, 'nc) fixed_mat
where
 plus_fixed_mat = fixed_mat_add

definition minus_fixed_mat :: ('a, 'nr::finite, 'nc::finite) fixed_mat ⇒ ('a, 'nr, 'nc) fixed_mat ⇒ ('a, 'nr, 'nc) fixed_mat
where
 minus_fixed_mat A B = fixed_mat_add A (fixed_mat_smult (-1) B)

definition uminus_fixed_mat :: ('a, 'nr::finite, 'nc::finite) fixed_mat ⇒ ('a, 'nr, 'nc) fixed_mat **where**
 uminus_fixed_mat A = fixed_mat_smult (-1) A

definition scaleR_fixed_mat :: real ⇒ ('a, 'nr::finite, 'nc::finite) fixed_mat ⇒ ('a, 'nr, 'nc) fixed_mat **where**
 scaleR_fixed_mat r A = fixed_mat_smult (of_real r) A

definition norm_fixed_mat :: ('a, 'nr::finite, 'nc::finite) fixed_mat ⇒ real **where**
 norm_fixed_mat A = sqrt (∑ i∈{o..<CARD('nr)}. ∑ j∈{o..<CARD('nc)}. (norm (fixed_mat_index A i j))^2)

definition dist_fixed_mat :: ('a, 'nr::finite, 'nc::finite) fixed_mat ⇒ ('a, 'nr, 'nc) fixed_mat ⇒ real **where**
 dist_fixed_mat A B = norm (A - B)

definition uniformity_fixed_mat :: ((('a::{real_algebra_1,real_normed_vector}, 'nr::finite, 'nc::finite) fixed_mat × ('a, 'nr, 'nc) fixed_mat) filter **where**
 uniformity_fixed_mat = (INF e∈{o<..}. principal {(x, y). dist x y < e})

definition open_fixed_mat :: ('a, 'nr::finite, 'nc::finite) fixed_mat set ⇒ bool **where**
 open_fixed_mat S = (∀ x∈S. ∀_F (x', y) in uniformity. x' = x → y ∈ S)

definition sgn_fixed_mat :: ('a, 'nr::finite, 'nc::finite) fixed_mat ⇒ ('a, 'nr, 'nc) fixed_mat **where**
 sgn_fixed_mat A = (if A = o then o
 else scaleR (1 / norm A) A)

lemma uminus_add: - (A :: ('a, 'nr::finite, 'nc::finite) fixed_mat) + A = o

proof -

have ao: Rep_fixed_mat A ∈ carrier_mat CARD('nr) CARD('nc)

using Rep_fixed_mat by blast

have a1: - (Rep_fixed_mat A) + (Rep_fixed_mat A) = o_m CARD('nr) CARD('nc)

using uminus_l_inv_mat[of Rep_fixed_mat A CARD('nr) CARD('nc)] ao by simp

have a2: - A + A = Abs_fixed_mat (Rep_fixed_mat (- A + A))

by (simp add: Rep_fixed_mat_inverse)

```

have a3: Rep_fixed_mat (- A + A) = Rep_fixed_mat (- A) + Rep_fixed_mat A
using Rep_fixed_mat_add[of -A A] unfolding fixed_mat_add_def
by (simp add: plus_fixed_mat_def)
have a4: Rep_fixed_mat (- A) = - Rep_fixed_mat A
using fixed_mat_smult.rep_eq[of -1 A]
unfolding uminus_fixed_mat_def fixed_mat_smult_def by auto
have a5: Abs_fixed_mat (om CARD('nr) CARD('nc)) = Abs_fixed_mat (Rep_fixed_mat (fixed_mat_zero:: ('a, 'nr::finite, 'nc::finite) fixed_mat))
using Rep_fixed_mat_zero unfolding fixed_mat_zero_def
by metis
have a6: Abs_fixed_mat (Rep_fixed_mat (fixed_mat_zero:: ('a, 'nr::finite, 'nc::finite) fixed_mat)) =
  (fixed_mat_zero:: ('a, 'nr::finite, 'nc::finite) fixed_mat)
using Rep_fixed_mat_inverse[of (fixed_mat_zero:: ('a, 'nr::finite, 'nc::finite) fixed_mat)] by simp
have a7: (fixed_mat_zero:: ('a, 'nr::finite, 'nc::finite) fixed_mat) = 0
unfolding zero_fixed_mat_def
by simp
show ?thesis using a0 a1 a2 a3 a4 a5 a6 a7
by (smt (verit, best))
qed

```

lemma smult: $a *_R b *_R x = (a * b) *_R (x:: ('a, 'nr::finite, 'nc::finite) fixed_mat)$

proof -

```

have b8: a *_R b *_R x = scaleR a (scaleR b x)
by (simp add: scaleR_fixed_mat_def)
also have b7: ... = fixed_mat_smult (of_real a) (fixed_mat_smult (of_real b) x)
by (simp add: scaleR_fixed_mat_def)
also have b6: ... = Abs_fixed_mat (of_real a ·m Rep_fixed_mat (fixed_mat_smult (of_real b) x))
by (simp add: fixed_mat_smult.rep_eq fixed_mat_smult_def)
also have b5: ... = Abs_fixed_mat (of_real a ·m (of_real b ·m Rep_fixed_mat x))
by (simp add: fixed_mat_smult.rep_eq)
also have b4: ... = Abs_fixed_mat ((of_real a * of_real b) ·m Rep_fixed_mat x)
using mult_smult_assoc_mat Rep_fixed_mat

```

proof -

```

have  $\forall i j. i < \text{CARD}('nr) \wedge j < \text{CARD}('nc) \longrightarrow$ 
  (of_real a ·m (of_real b ·m Rep_fixed_mat x)) $$ (i, j) =
  ((of_real a * of_real b) ·m Rep_fixed_mat x) $$ (i, j)

```

proof -

```

have a0:  $\forall i j. i < \text{CARD}('nr) \wedge j < \text{CARD}('nc) \longrightarrow$ 
  (of_real a ·m (of_real b ·m Rep_fixed_mat x)) $$ (i, j) =
  of_real a * ((of_real b ·m Rep_fixed_mat x) $$ (i, j))
unfolding smult_mat_def
by (metis Rep_fixed_mat carrier_matD(1) carrier_matD(2) fixed_mat_smult.rep_eq index_smult_mat(1)
smult_mat_def)

```

```

have a1:  $\forall i j. i < \text{CARD}('nr) \wedge j < \text{CARD}('nc) \longrightarrow$ 
  of_real a * ((of_real b ·m Rep_fixed_mat x) $$ (i, j)) =
  of_real a * (of_real b * (Rep_fixed_mat x $$ (i, j)))
unfolding smult_mat_def index_mat_def using index_smult_mat(1)
by (metis Rep_fixed_mat carrier_matD(1) carrier_matD(2) index_mat_def smult_mat_def)
have a2:  $\forall i j. i < \text{CARD}('nr) \wedge j < \text{CARD}('nc) \longrightarrow$ 
  of_real a * (of_real b * (Rep_fixed_mat x $$ (i, j)))
  = (of_real a * of_real b) * (Rep_fixed_mat x $$ (i, j))
by (simp add: mult.assoc)
have a3:  $\forall i j. i < \text{CARD}('nr) \wedge j < \text{CARD}('nc) \longrightarrow$ 

```

```

    (of_real a * of_real b) * (Rep_fixed_mat x $$$ (i, j))
  = ((of_real a * of_real b) ·m Rep_fixed_mat x) $$$ (i, j)
  unfolding smult_mat_def index_mat_def using index_smult_mat(1)
  by (metis Rep_fixed_mat carrier_matD(1) carrier_matD(2) index_mat_def smult_mat_def)
show ?thesis using a0 a1 a2 a3 by presburger
qed
hence of_real a ·m (of_real b ·m Rep_fixed_mat x) =
  (of_real a * of_real b) ·m Rep_fixed_mat x
using mat_eq_iff Rep_fixed_mat unfolding carrier_mat_def
by (smt (verit, best) Rep_fixed_mat carrier_matD(1) carrier_matD(2) index_smult_mat(2) index_smult_mat(3))
thus ?thesis by simp
qed
also have b1: ... = Abs_fixed_mat (of_real (a * b) ·m Rep_fixed_mat x)
  by simp
also have b0: ... = fixed_mat_smult (of_real (a * b)) x
  by (simp add: fixed_mat_smult.rep_eq fixed_mat_smult_def)
also have b3: ... = scaleR(a * b) x
  by (simp add: scaleR_fixed_mat_def)
also have b2: ... = (a * b) *R x
  by (simp add: scaleR_fixed_mat_def)
show ?thesis
  using b0 b1 b2 b3 b4 b5 b6 b7 b8
  by argo
qed

lemma scaleR: 1 *R x = (x :: ('a, 'nr::finite, 'nc::finite) fixed_mat)
proof -
  have a0: 1 *R x = scaleR 1 x
    by (simp add: scaleR_fixed_mat_def)
  have a1: ... = fixed_mat_smult (of_real 1) x
    by (simp add: scaleR_fixed_mat_def)
  have a2: ... = fixed_mat_smult 1 x
    by simp
  have a3: ... = x
  proof -
    have fixed_mat_smult 1 x = Abs_fixed_mat (smult_mat 1 (Rep_fixed_mat x))
      by (simp add: fixed_mat_smult_def)
    also have smult_mat 1 (Rep_fixed_mat x) = Rep_fixed_mat x
    proof -
      have ∀ i j. i < dim_row (Rep_fixed_mat x) ∧ j < dim_col (Rep_fixed_mat x) ⟶
        (smult_mat 1 (Rep_fixed_mat x)) $$$ (i, j) = Rep_fixed_mat x $$$ (i, j)
        by (auto simp add: smult_mat_def)
      thus ?thesis by auto
    qed
  qed
  hence Abs_fixed_mat (smult_mat 1 (Rep_fixed_mat x)) = Abs_fixed_mat (Rep_fixed_mat x) by simp
  also have ... = x by (rule Rep_fixed_mat_inverse)
  finally show fixed_mat_smult 1 x = x .
qed
show 1 *R x = x
  using a0 a1 a2 a3 by simp
qed

```

```

lemma scaleR_0: 0 *R x = (0 :: ('a, 'nr::finite, 'nc::finite) fixed_mat)

```

```

proof –
  have a0:  $o *_{\mathbb{R}} x = \text{scaleR } o \ x$ 
    by (simp add: scaleR_fixed_mat_def)
  have a1:  $\dots = \text{fixed\_mat\_smult } (\text{of\_real } o) \ x$ 
    by (simp add: scaleR_fixed_mat_def)
  have a2:  $\dots = \text{fixed\_mat\_smult } o \ x$ 
    by simp
  have a3:  $\dots = o$ 
proof –
  have fixed_mat_smult o x = Abs_fixed_mat (smult_mat o (Rep_fixed_mat x))
    by (simp add: fixed_mat_smult_def)
  also have smult_mat o (Rep_fixed_mat x) = (Rep_fixed_mat (o :: ('a, 'nr)::finite, 'nc)::finite) fixed_mat)
proof –
  have  $o \cdot_m \text{Rep\_fixed\_mat } x = o_m (\text{CARD}('nr)) (\text{CARD}('nc))$ 
proof –
  have  $\forall i j. i < \text{dim\_row } (\text{Rep\_fixed\_mat } x) \wedge j < \text{dim\_col } (\text{Rep\_fixed\_mat } x) \longrightarrow$ 
     $(o \cdot_m \text{Rep\_fixed\_mat } x) \ \$\$ \ (i, j) = o$ 
    unfolding smult_mat_def by auto
  then have  $o \cdot_m \text{Rep\_fixed\_mat } x = \text{Matrix.mat } \text{CARD}('nr) \ \text{CARD}('nc) \ (\lambda ij. o)$ 
    using Rep_fixed_mat[of x] by auto
  moreover have  $\text{Matrix.mat } \text{CARD}('nr) \ \text{CARD}('nc) \ (\lambda ij. o) = o_m (\text{CARD}('nr)) (\text{CARD}('nc))$ 
    by (simp add: zero_mat_def)
  ultimately show ?thesis by simp
  qed
  also have  $\dots = \text{Rep\_fixed\_mat } (\text{fixed\_mat\_zero} :: ('a, 'nr)::finite, 'nc)::finite) \ \text{fixed\_mat}$ 
    apply(subst fixed_mat_zero_def)
    by (metis Rep_fixed_mat_inverse Rep_fixed_mat_zero)
  finally show ?thesis unfolding zero_fixed_mat_def by blast
  qed
  hence Abs_fixed_mat (smult_mat o (Rep_fixed_mat x)) = Abs_fixed_mat (Rep_fixed_mat (o :: ('a, 'nr)::finite, 'nc)::finite) fixed_mat)
by simp
  also have  $\dots = (o :: ('a, 'nr)::finite, 'nc)::finite) \ \text{fixed\_mat}$ 
    using Rep_fixed_mat_inverse by auto
  finally show fixed_mat_smult o x = (o :: ('a, 'nr)::finite, 'nc)::finite) fixed_mat) .
qed
  show  $o *_{\mathbb{R}} x = o$ 
    using a0 a1 a2 a3 by simp
qed

```

```

lemma norm_o: norm (o :: ('a, 'nr)::finite, 'nc)::finite) fixed_mat) = o
proof –
  have a0:  $\forall i \in \{0..<\text{CARD}('nr)\}. \forall j \in \{0..<\text{CARD}('nc)\}.$ 
     $\text{fixed\_mat\_index } (o :: ('a, 'nr, 'nc) \ \text{fixed\_mat}) \ i \ j = o$ 
    unfolding zero_fixed_mat_def fixed_mat_zero_def zero_mat_def fixed_mat_index_def
    by (simp add: Abs_fixed_mat_inverse)
  have norm (o :: ('a, 'nr, 'nc) fixed_mat) =
     $\text{sqrt } (\sum i \in \{0..<\text{CARD}('nr)\}. \sum j \in \{0..<\text{CARD}('nc)\}.$ 
     $(\text{norm } (\text{fixed\_mat\_index } (o :: ('a, 'nr, 'nc) \ \text{fixed\_mat}) \ i \ j))^2)$ 
    by (simp add: norm_fixed_mat_def)
  also have  $\dots = \text{sqrt } (\sum i \in \{0..<\text{CARD}('nr)\}. \sum j \in \{0..<\text{CARD}('nc)\}. o^2)$ 
    using a0 by simp
  also have  $\dots = \text{sqrt } o$  by simp
  finally show ?thesis by simp

```

qed

lemma sgn: $\text{sgn } x = \text{inverse } (\text{norm } x) *_{\mathbb{R}} (x :: ('a, 'nr::\text{finite}, 'nc::\text{finite}) \text{ fixed_mat})$

proof –

have $\text{sgn } x = (\text{if } x = 0 \text{ then } 0 \text{ else } \text{scaleR } (1 / \text{norm } x) x)$

by (*simp add: sgn_fixed_mat_def*)

show $\text{sgn } x = \text{inverse } (\text{norm } x) *_{\mathbb{R}} x$

proof (*cases x = 0*)

case True

have $\text{inverse } (\text{norm } x) *_{\mathbb{R}} x = \text{inverse } (\text{norm } (0 :: ('a, 'nr::\text{finite}, 'nc::\text{finite}) \text{ fixed_mat})) *_{\mathbb{R}} 0$

using True by simp

also have $\dots = \text{inverse } 0 *_{\mathbb{R}} 0$

using norm_0 using arg_cong by fast

also have $\dots = 0$

using scaleR_0 by simp

finally have $\text{inverse } (\text{norm } x) *_{\mathbb{R}} x = 0$.

with True show ?thesis

by (*simp add: sgn_fixed_mat_def*)

next

case False

have $b_0: \text{sgn } x = \text{scaleR } (1 / \text{norm } x) x$

using False by (*simp add: sgn_fixed_mat_def*)

have $b_1: \dots = \text{scaleR } (\text{inverse } (\text{norm } x)) x$

by (*simp add: divide_inverse*)

have $b_2: \dots = \text{inverse } (\text{norm } x) *_{\mathbb{R}} x$

by (*simp add: scaleR_fixed_mat_def*)

show ?thesis using b0 b1 b2 by simp

qed

qed

lemma norm_eq_zero_iff: $(\text{norm } x = (0 :: \text{real})) = (x = (0 :: ('a, 'nr::\text{finite}, 'nc::\text{finite}) \text{ fixed_mat}))$

proof

assume $\text{norm } x = 0$

have $\text{norm_def: norm } x = \text{sqrt } (\sum_{i \in \{0..<\text{CARD}('nr)\}} \sum_{j \in \{0..<\text{CARD}('nc)\}} (\text{norm } (\text{fixed_mat_index } x \ i \ j))^2)$

by (*simp add: norm_fixed_mat_def*)

with $\langle \text{norm } x = 0 \rangle$ **have** $b_0: \text{sqrt } (\sum_{i \in \{0..<\text{CARD}('nr)\}} \sum_{j \in \{0..<\text{CARD}('nc)\}} (\text{norm } (\text{fixed_mat_index } x \ i \ j))^2) =$

0

by simp

then have $b_1: (\sum_{i \in \{0..<\text{CARD}('nr)\}} \sum_{j \in \{0..<\text{CARD}('nc)\}} (\text{norm } (\text{fixed_mat_index } x \ i \ j))^2) = 0$

by simp

then have $b_2: \forall i \in \{0..<\text{CARD}('nr)\}. \forall j \in \{0..<\text{CARD}('nc)\}. (\text{norm } (\text{fixed_mat_index } x \ i \ j))^2 = 0$

proof –

have $\text{non_neg: } \bigwedge i \ j. (\text{norm } (\text{fixed_mat_index } x \ i \ j))^2 \geq 0$

by (*simp add: power2_eq_square*)

{

fix $i \ j$

assume $i_range: i \in \{0..<\text{CARD}('nr)\}$

assume $j_range: j \in \{0..<\text{CARD}('nc)\}$

have $(\text{norm } (\text{fixed_mat_index } x \ i \ j))^2 \leq (\sum_{i = 0..<\text{CARD}('nr)}. \sum_{j = 0..<\text{CARD}('nc)}. (\text{norm } (\text{fixed_mat_index } x \ i \ j))^2)$

proof –

have $a_0: (\text{norm } (\text{fixed_mat_index } x \ i \ j))^2 = (\sum_{j' = 0..<\text{CARD}('nc)}. \text{if } j' = j \text{ then } (\text{norm } (\text{fixed_mat_index } x \ i \ j))^2$
else 0)

using j_range by (*simp add: sum.delta*)

```

also have ... ≤ (∑ j' = 0..<CARD('nc). (norm (fixed_mat_index x i j'))2)
proof (rule sum_mono)
  fix j'
  assume j' ∈ {0..<CARD('nc)}
  show (if j' = j then (norm (fixed_mat_index x i j))2 else 0) ≤ (norm (fixed_mat_index x i j'))2
  proof (cases j' = j)
    case True
    then show ?thesis by simp
  next
  case False
  have 0 ≤ (norm (fixed_mat_index x i j'))2 by (simp add: non_neg)
  show ?thesis by simp
qed
qed
have a1: ... = (∑ i' = 0..<CARD('nr). ∑ j' = 0..<CARD('nc).
  if i' = i then (norm (fixed_mat_index x i j'))2 else 0)
  using i_range sum.delta[of {0..<CARD('nr)} i λ j'. (norm (fixed_mat_index x i j'))2]
  proof -
  have (∑ i' = 0..<CARD('nr). ∑ j' = 0..<CARD('nc). if i' = i then (norm (fixed_mat_index x i j'))2 else 0) =
    (∑ i' = 0..<CARD('nr). if i' = i then (∑ j' = 0..<CARD('nc). (norm (fixed_mat_index x i j'))2 else 0)
    by (simp add: sum.swap)
  also have ... = (if i ∈ {0..<CARD('nr)} then (∑ j' = 0..<CARD('nc). (norm (fixed_mat_index x i j'))2 else 0)
  by simp
  also have ... = (∑ j' = 0..<CARD('nc). (norm (fixed_mat_index x i j'))2)
  using i_range by simp
  finally show (∑ j' = 0..<CARD('nc). (norm (fixed_mat_index x i j'))2) =
    (∑ i' = 0..<CARD('nr). ∑ j' = 0..<CARD('nc). if i' = i then (norm (fixed_mat_index x i j'))2 else 0)
  by simp
qed
have a2: ... ≤ (∑ i' = 0..<CARD('nr). ∑ j' = 0..<CARD('nc). (norm (fixed_mat_index x i' j'))2)
proof (rule sum_mono)
  fix i'
  assume i' ∈ {0..<CARD('nr)}
  show (∑ j' = 0..<CARD('nc). if i' = i then (norm (fixed_mat_index x i j'))2 else 0) ≤
    (∑ j' = 0..<CARD('nc). (norm (fixed_mat_index x i' j'))2)
  proof (cases i' = i)
    case True
    then show ?thesis by simp
  next
  case False
  have c0: (∑ j' = 0..<CARD('nc). if i' = i then (norm (fixed_mat_index x i j'))2 else 0) = 0
  using False by simp
  have c1: ... ≤ (∑ j' = 0..<CARD('nc). (norm (fixed_mat_index x i' j'))2)
  by (simp add: sum_nonneg non_neg)
  show ?thesis using c0 c1 by simp
qed
qed
show ?thesis using a0 a1 a2
  using <(∑ j' = 0..<CARD('nc). if j' = j then (norm (fixed_mat_index x i j))2 else 0) ≤ (∑ j' = 0..<CARD('nc).
  (norm (fixed_mat_index x i j'))2> by linarith
qed
with non_neg[of i j] have (norm (fixed_mat_index x i j))2 = 0
  using <(∑ i = 0..<CARD('nr). ∑ j = 0..<CARD('nc). (norm (fixed_mat_index x i j))2) = 0> by linarith
}

```

```

thus b3:  $\forall i \in \{0..<CARD('nr)\}. \forall j \in \{0..<CARD('nc)\}. (norm (fixed\_mat\_index\ x\ i\ j))^2 = 0$ 
by blast
qed
then have b4:  $\forall i \in \{0..<CARD('nr)\}. \forall j \in \{0..<CARD('nc)\}. norm (fixed\_mat\_index\ x\ i\ j) = 0$ 
by (simp add: power2_eq_iff)
then have b5:  $\forall i \in \{0..<CARD('nr)\}. \forall j \in \{0..<CARD('nc)\}. fixed\_mat\_index\ x\ i\ j = 0$ 
by simp
then have x = 0
proof –
have  $\forall i\ j. i < CARD('nr) \wedge j < CARD('nc) \longrightarrow fixed\_mat\_index\ x\ i\ j = 0$ 
using  $\langle \forall i \in \{0..<CARD('nr)\}. \forall j \in \{0..<CARD('nc)\}. fixed\_mat\_index\ x\ i\ j = 0 \rangle$ 
by auto
have x = Abs_fixed_mat (Rep_fixed_mat x)
by (simp add: Rep_fixed_mat_inverse)
also have Rep_fixed_mat x = Matrix.mat CARD('nr) CARD('nc) ( $\lambda(i,j). fixed\_mat\_index\ x\ i\ j$ )
using Rep_fixed_mat unfolding fixed_mat_index_def carrier_mat_def
by force
also have ... = Matrix.mat CARD('nr) CARD('nc) ( $\lambda(i,j). 0$ )
using  $\langle \forall i\ j. i < CARD('nr) \wedge j < CARD('nc) \longrightarrow fixed\_mat\_index\ x\ i\ j = 0 \rangle$ 
by (simp add: mat_eq_iff)
also have ... =  $o_m\ CARD('nr)\ CARD('nc)$ 
unfolding zero_mat_def by auto
also have Abs_fixed_mat ( $o_m\ CARD('nr)\ CARD('nc)$ ) = (fixed_mat_zero::('a, 'nr::finite, 'nc::finite) fixed_mat)
unfolding fixed_mat_zero_def by simp
also have fixed_mat_zero = 0
using zero_fixed_mat_def by metis
show x = 0
by (simp add: calculation zero_fixed_mat_def)
qed
thus x = 0 by simp
next
assume x = 0
have norm_def: norm x = sqrt ( $\sum i \in \{0..<CARD('nr)\}. \sum j \in \{0..<CARD('nc)\}. (norm (fixed\_mat\_index\ x\ i\ j))^2$ )
by (simp add: norm_fixed_mat_def)
from  $\langle x = 0 \rangle$  have  $\forall i \in \{0..<CARD('nr)\}. \forall j \in \{0..<CARD('nc)\}. fixed\_mat\_index\ x\ i\ j = 0$ 
proof –
have  $\forall i\ j. i < CARD('nr) \wedge j < CARD('nc) \longrightarrow fixed\_mat\_index\ x\ i\ j = 0$ 
by (simp add: NN_Lipschitz_Continuous.zero_fixed_mat_def  $\langle (x::('a, 'nr, 'nc) fixed\_mat) = (o::('a, 'nr, 'nc) fixed\_mat) \rangle$  fixed_mat_index.rep_eq)
have x = Abs_fixed_mat (Rep_fixed_mat x)
by (simp add: Rep_fixed_mat_inverse)
also have Rep_fixed_mat x = Matrix.mat CARD('nr) CARD('nc) ( $\lambda(i,j). fixed\_mat\_index\ x\ i\ j$ )
using Rep_fixed_mat unfolding fixed_mat_index_def carrier_mat_def
by force
also have ... = Matrix.mat CARD('nr) CARD('nc) ( $\lambda(i,j). 0$ )
using  $\langle \forall i\ j. i < CARD('nr) \wedge j < CARD('nc) \longrightarrow fixed\_mat\_index\ x\ i\ j = 0 \rangle$ 
by (simp add: mat_eq_iff)
also have ... =  $o_m\ CARD('nr)\ CARD('nc)$ 
unfolding zero_mat_def by auto
also have Abs_fixed_mat ( $o_m\ CARD('nr)\ CARD('nc)$ ) = (fixed_mat_zero::('a, 'nr::finite, 'nc::finite) fixed_mat)
unfolding fixed_mat_zero_def by simp
also have fixed_mat_zero = 0
using zero_fixed_mat_def by metis
show  $\forall i \in \{0..<CARD('nr)\}. \forall j \in \{0..<CARD('nc)\}. fixed\_mat\_index\ x\ i\ j = 0$ 

```

using $\langle \forall (i::\text{nat}) j::\text{nat}. i < \text{CARD}('nr) \wedge j < \text{CARD}('nc) \longrightarrow \text{fixed_mat_index } (x::('a, 'nr, 'nc) \text{ fixed_mat}) i j = (o::'a) \rangle$
atLeastoLessThan **by** **blast**
qed
then have $\forall i \in \{o..<\text{CARD}('nr)\}. \forall j \in \{o..<\text{CARD}('nc)\}. \text{norm } (\text{fixed_mat_index } x i j) = o$
by **simp**
then have $\forall i \in \{o..<\text{CARD}('nr)\}. \forall j \in \{o..<\text{CARD}('nc)\}. (\text{norm } (\text{fixed_mat_index } x i j))^2 = o$
by **simp**
then have $(\sum i \in \{o..<\text{CARD}('nr)\}. \sum j \in \{o..<\text{CARD}('nc)\}. (\text{norm } (\text{fixed_mat_index } x i j))^2) = o$
by **simp**
then have **sqrt** $(\sum i \in \{o..<\text{CARD}('nr)\}. \sum j \in \{o..<\text{CARD}('nc)\}. (\text{norm } (\text{fixed_mat_index } x i j))^2) = o$
by **simp**
with **norm_def** **show** $\text{norm } x = o$
by **simp**
qed

lemma **sum_tuple**: $\langle (\sum i < n. \sum j < m. P i j) = (\sum p \in \{(i,j). i < n \wedge j < m\}. P (\text{fst } p) (\text{snd } p)) \rangle$
apply(**subst prod.split_sel**, **simp**)
apply(**subst sum.cartesian_product'**)
by(**simp**, **metis lessThan_def**)

lemma **triangle_inequality**: $\text{norm } ((x::('a, 'nr::\text{finite}, 'nc::\text{finite}) \text{ fixed_mat}) + y::('a, 'nr::\text{finite}, 'nc::\text{finite}) \text{ fixed_mat}) \leq \text{norm } x + \text{norm } y$

proof –

have **norm_def**: $\text{norm } z = \text{sqrt } (\sum i \in \{o..<\text{CARD}('nr)\}. \sum j \in \{o..<\text{CARD}('nc)\}. (\text{norm } (\text{fixed_mat_index } z i j))^2)$
for $z :: ('a, 'nr::\text{finite}, 'nc::\text{finite}) \text{ fixed_mat}$
by (**simp add**: **norm_fixed_mat_def**)
have **component_add**: $\text{fixed_mat_index } (x + y) i j = \text{fixed_mat_index } x i j + \text{fixed_mat_index } y i j$
if $i \in \{o..<\text{CARD}('nr)\} j \in \{o..<\text{CARD}('nc)\}$ **for** $i j$
proof –
have $\text{fixed_mat_index } (x + y) i j = \text{Rep_fixed_mat } (x + y) \text{ \&\amp; } (i, j)$
by (**simp add**: **fixed_mat_index_def**)
also have $\text{Rep_fixed_mat } (x + y) \text{ \&\amp; } (i, j) = \text{Rep_fixed_mat } ((\text{Abs_fixed_mat } (\text{Rep_fixed_mat } x + \text{Rep_fixed_mat } y))::('a, 'nr::\text{finite}, 'nc::\text{finite}) \text{ fixed_mat}) \text{ \&\& } (i::\text{nat}, j)$
unfolding **plus_fixed_mat_def** **fixed_mat_add_def** **plus_mat_def** **by** **blast**
have $\text{Rep_fixed_mat } x \in \text{carrier_mat } \text{CARD}('nr) \text{ CARD}('nc)$
by (**simp add**: **Rep_fixed_mat**)
moreover have $\text{Rep_fixed_mat } y \in \text{carrier_mat } \text{CARD}('nr) \text{ CARD}('nc)$
by (**simp add**: **Rep_fixed_mat**)
ultimately have $\text{ao: Rep_fixed_mat } x + \text{Rep_fixed_mat } y \in \text{carrier_mat } \text{CARD}('nr) \text{ CARD}('nc)$
by **auto**
hence $\text{Rep_fixed_mat } (\text{Abs_fixed_mat } (\text{Rep_fixed_mat } x + \text{Rep_fixed_mat } y)::('a, 'nr::\text{finite}, 'nc::\text{finite}) \text{ fixed_mat}) = \text{Rep_fixed_mat } x + \text{Rep_fixed_mat } y$
using **Abs_fixed_mat_inverse**[**of** $\text{Rep_fixed_mat } x + \text{Rep_fixed_mat } y$] **by** **blast**
hence $\text{Rep_fixed_mat } ((\text{Abs_fixed_mat } (\text{Rep_fixed_mat } x + \text{Rep_fixed_mat } y))::('a, 'nr::\text{finite}, 'nc::\text{finite}) \text{ fixed_mat}) \text{ \&\& } (i, j) =$
 $(\text{Rep_fixed_mat } x + \text{Rep_fixed_mat } y) \text{ \&\& } (i, j)$
by **metis**
also have $\dots = \text{Rep_fixed_mat } x \text{ \&\& } (i, j) + \text{Rep_fixed_mat } y \text{ \&\& } (i, j)$
proof –
have **add_def**:
 $\text{Rep_fixed_mat } (\text{fixed_mat_add } x y) = \text{Rep_fixed_mat } x + \text{Rep_fixed_mat } y$

```

using fixed_mat_add_def Abs_fixed_mat_inverse by auto
have index_def:
  (Rep_fixed_mat x + Rep_fixed_mat y) $$ (i, j) = (Rep_fixed_mat x) $$ (i, j) + (Rep_fixed_mat y) $$ (i, j)
proof –
  have carrier_x: Rep_fixed_mat x ∈ carrier_mat CARD('nr) CARD('nc)
    by (simp add: Rep_fixed_mat)
  have carrier_y: Rep_fixed_mat y ∈ carrier_mat CARD('nr) CARD('nc)
    by (simp add: Rep_fixed_mat)
  have add_def: Rep_fixed_mat x + Rep_fixed_mat y =
    mat CARD('nr) CARD('nc) (λ(i', j'). Rep_fixed_mat x $$ (i', j') + Rep_fixed_mat y $$ (i', j'))
  using carrier_x carrier_y unfolding plus_mat_def mat_def carrier_mat_def
  by (simp add: cond_case_prod_eta)
  have (Rep_fixed_mat x + Rep_fixed_mat y) $$ (i, j) =
    mat CARD('nr) CARD('nc) (λ(i', j'). Rep_fixed_mat x $$ (i', j') + Rep_fixed_mat y $$ (i', j')) $$ (i, j)
  by (simp add: add_def)
  moreover have mat CARD('nr) CARD('nc) (λ(i', j'). Rep_fixed_mat x $$ (i', j') + Rep_fixed_mat y $$ (i', j')) $$
(i, j) =
  (λ(i', j'). Rep_fixed_mat x $$ (i', j') + Rep_fixed_mat y $$ (i', j')) (i, j)
  using Rep_fixed_mat[of x] Rep_fixed_mat[of y] unfolding index_mat_def carrier_mat_def
  by (metis (no_types, lifting) atLeastLessThan_iff index_mat(1) index_mat_def that(1) that(2))
  moreover have (λ(i', j'). Rep_fixed_mat x $$ (i', j') + Rep_fixed_mat y $$ (i', j')) (i, j) =
    Rep_fixed_mat x $$ (i, j) + Rep_fixed_mat y $$ (i, j)
  by simp
  show ?thesis
    by (simp add: calculation(2) local.add_def)
qed
show ?thesis
  using add_def index_def
  by simp
qed
also have ... = fixed_mat_index x i j + fixed_mat_index y i j
  by (simp add: fixed_mat_index_def)
finally show fixed_mat_index (x + y) i j = fixed_mat_index x i j + fixed_mat_index y i j
  by (simp add: fixed_mat_add_def fixed_mat_index.rep_eq plus_fixed_mat_def)
qed
have norm (fixed_mat_index (x + y) i j) ≤ norm (fixed_mat_index x i j) + norm (fixed_mat_index y i j)
  if i ∈ {0..<CARD('nr)} j ∈ {0..<CARD('nc)} for i j
  using component_add norm_triangle_ineq that by auto
then have component_ineq: (norm (fixed_mat_index (x + y) i j))^2 ≤
  (norm (fixed_mat_index x i j) + norm (fixed_mat_index y i j))^2
  if i ∈ {0..<CARD('nr)} j ∈ {0..<CARD('nc)} for i j
  using that by simp
have expand_square: (a + b)^2 = a^2 + 2*a*b + b^2 for a b :: real
  by (simp add: power2_eq_square algebra_simps)
have component_ineq2: (norm (fixed_mat_index (x + y) i j))^2 ≤
  (norm (fixed_mat_index x i j))^2 + 2*(norm (fixed_mat_index x i j))*(norm (fixed_mat_index y i j)) +
  (norm (fixed_mat_index y i j))^2
  if i ∈ {0..<CARD('nr)} j ∈ {0..<CARD('nc)} for i j
  using component_ineq expand_square that by simp
have sum_ineq: (∑ i ∈ {0..<CARD('nr)}. ∑ j ∈ {0..<CARD('nc)}. (norm (fixed_mat_index (x + y) i j))^2) ≤
  (∑ i ∈ {0..<CARD('nr)}. ∑ j ∈ {0..<CARD('nc)}. ((norm (fixed_mat_index x i j))^2) +
  2*(norm (fixed_mat_index x i j))*(norm (fixed_mat_index y i j)) +
  (norm (fixed_mat_index y i j))^2)
proof –

```

```

have  $\forall i \in \{0..<CARD('nr)\}. \forall j \in \{0..<CARD('nc)\}.$ 
  (norm (fixed_mat_index (x + y) i j))^2 ≤
  (norm (fixed_mat_index x i j))^2 + 2*(norm (fixed_mat_index x i j))*(norm (fixed_mat_index y i j)) +
  (norm (fixed_mat_index y i j))^2
using component_ineq2 by blast
have inner_sum_mono:  $\forall i \in \{0..<CARD('nr)\}.$ 
  ( $\sum j \in \{0..<CARD('nc)\}.$  (norm (fixed_mat_index (x + y) i j))^2) ≤
  ( $\sum j \in \{0..<CARD('nc)\}.$  (norm (fixed_mat_index x i j))^2 +
  2 * norm (fixed_mat_index x i j) * norm (fixed_mat_index y i j) +
  (norm (fixed_mat_index y i j))^2)
proof
fix i
assume i ∈ {0..<CARD('nr)}
have  $\forall j \in \{0..<CARD('nc)\}.$ 
  (norm (fixed_mat_index (x + y) i j))^2 ≤
  (norm (fixed_mat_index x i j))^2 +
  2 * norm (fixed_mat_index x i j) * norm (fixed_mat_index y i j) +
  (norm (fixed_mat_index y i j))^2
using <i ∈ {0..<CARD('nr)}> component_ineq
using component_ineq2 by blast
thus ( $\sum j \in \{0..<CARD('nc)\}.$  (norm (fixed_mat_index (x + y) i j))^2) ≤
  ( $\sum j \in \{0..<CARD('nc)\}.$  (norm (fixed_mat_index x i j))^2 +
  2 * norm (fixed_mat_index x i j) * norm (fixed_mat_index y i j) +
  (norm (fixed_mat_index y i j))^2)
using sum_mono by fast
qed
have ( $\sum i \in \{0..<CARD('nr)\}.$   $\sum j \in \{0..<CARD('nc)\}.$  (norm (fixed_mat_index (x + y) i j))^2) ≤
  ( $\sum i \in \{0..<CARD('nr)\}.$   $\sum j \in \{0..<CARD('nc)\}.$  (norm (fixed_mat_index x i j))^2 +
  2 * norm (fixed_mat_index x i j) * norm (fixed_mat_index y i j) +
  (norm (fixed_mat_index y i j))^2)
using inner_sum_mono sum_mono by fast
thus ?thesis .
qed
have sum_distribute: ( $\sum i \in \{0..<CARD('nr)\}.$   $\sum j \in \{0..<CARD('nc)\}.$  (norm (fixed_mat_index x i j))^2 +
  2*(norm (fixed_mat_index x i j))*(norm (fixed_mat_index y i j)) +
  (norm (fixed_mat_index y i j))^2) =
  ( $\sum i \in \{0..<CARD('nr)\}.$   $\sum j \in \{0..<CARD('nc)\}.$  (norm (fixed_mat_index x i j))^2) +
  ( $\sum i \in \{0..<CARD('nr)\}.$   $\sum j \in \{0..<CARD('nc)\}.$  2*(norm (fixed_mat_index x i j))*(norm (fixed_mat_index y i j)))
+
  ( $\sum i \in \{0..<CARD('nr)\}.$   $\sum j \in \{0..<CARD('nc)\}.$  (norm (fixed_mat_index y i j))^2)
by (simp add: sum.distrib)
have cs: ( $\sum i \in \{0..<CARD('nr)\}.$   $\sum j \in \{0..<CARD('nc)\}.$  2*(norm (fixed_mat_index x i j))*(norm (fixed_mat_index y i j))) ≤
  2 * sqrt( $\sum i \in \{0..<CARD('nr)\}.$   $\sum j \in \{0..<CARD('nc)\}.$  (norm (fixed_mat_index x i j))^2) *
  sqrt( $\sum i \in \{0..<CARD('nr)\}.$   $\sum j \in \{0..<CARD('nc)\}.$  (norm (fixed_mat_index y i j))^2)
proof —
let ?x_norm_ij =  $\lambda i j.$  norm (fixed_mat_index x i j)
let ?y_norm_ij =  $\lambda i j.$  norm (fixed_mat_index y i j)
let ?sum_xy =  $\sum i = 0..<CARD('nr).$   $\sum j = 0..<CARD('nc).$  ?x_norm_ij i j * ?y_norm_ij i j
let ?sum_x2 =  $\sum i = 0..<CARD('nr).$   $\sum j = 0..<CARD('nc).$  (?x_norm_ij i j)^2
let ?sum_y2 =  $\sum i = 0..<CARD('nr).$   $\sum j = 0..<CARD('nc).$  (?y_norm_ij i j)^2
have step1: ( $\sum i = 0..<CARD('nr).$   $\sum j = 0..<CARD('nc).$  2 * ?x_norm_ij i j * ?y_norm_ij i j) =
  2 * ( $\sum i = 0..<CARD('nr).$   $\sum j = 0..<CARD('nc).$  ?x_norm_ij i j * ?y_norm_ij i j)
using sum_distrib_left[of 2 _ {0..<CARD('nr)}]

```

```

proof –
  let ?g = λi. ∑ j = 0..<CARD('nc). norm (fixed_mat_index x i j) * norm (fixed_mat_index y i j)
  have 2 * (∑ i = 0..<CARD('nr). ?g i) = (∑ i = 0..<CARD('nr). 2 * ?g i)
    using sum_distrib_left[of 2 ?g {0..<CARD('nr)}] by blast
  have (∑ i = 0..<CARD('nr). 2 * (∑ j = 0..<CARD('nc). norm (fixed_mat_index x i j) * norm (fixed_mat_index y i
j))) =
    (∑ i = 0..<CARD('nr). (∑ j = 0..<CARD('nc). 2 * norm (fixed_mat_index x i j) * norm (fixed_mat_index y i
j)))
  proof –
    have ∧ i. 2 * (∑ j = 0..<CARD('nc). norm (fixed_mat_index x i j) * norm (fixed_mat_index y i j)) =
      (∑ j = 0..<CARD('nc). 2 * norm (fixed_mat_index x i j) * norm (fixed_mat_index y i j))
      using sum_distrib_left[of 2 _ {0..<CARD('nc)}]
      by (metis (mono_tags, lifting) Finite_Cartesian_Product.sum_cong_aux more_arith_simps(11))
    thus ?thesis by simp
  qed
  show ?thesis using <2 * (∑ i = 0..<CARD('nr). ?g i) = (∑ i = 0..<CARD('nr). 2 * ?g i)>
    and <(∑ i = 0..<CARD('nr). 2 * (∑ j = 0..<CARD('nc). norm (fixed_mat_index x i j) * norm (fixed_mat_index
y i j))) =
      (∑ i = 0..<CARD('nr). (∑ j = 0..<CARD('nc). 2 * norm (fixed_mat_index x i j) * norm (fixed_mat_index
y i j)))>
    by simp
  qed
  have cs_ineq: ?sum_xy^2 ≤ ?sum_x2 * ?sum_y2
  proof –
    let ?all_indices = {(i, j). i < CARD('nr) ∧ j < CARD('nc)}
    let ?x_vec = λ(i, j). ?x_norm_ij i j
    let ?y_vec = λ(i, j). ?y_norm_ij i j
    have rewrite_sum_xy: ?sum_xy = (∑ p ∈ ?all_indices. ?x_vec p * ?y_vec p)
      apply (subst case_prod_beta) +
      apply (subst sum_tuple[of λ i j . norm (fixed_mat_index x i j) * norm (fixed_mat_index y i j) CARD('nc) CARD('nr),
symmetric])
      using atLeast0LessThan by presburger
    have rewrite_sum_x2: ?sum_x2 = (∑ p ∈ ?all_indices. (?x_vec p)^2)
      apply (subst case_prod_beta)
      apply (subst sum_tuple[of λ i j . (norm (fixed_mat_index x i j))^2 CARD('nc) CARD('nr), symmetric])
      using atLeast0LessThan by presburger
    have rewrite_sum_y2: ?sum_y2 = (∑ p ∈ ?all_indices. (?y_vec p)^2)
      apply (subst case_prod_beta)
      using sum_tuple[of λ i j . (norm (fixed_mat_index y i j))^2 CARD('nc) CARD('nr), symmetric]
      using atLeast0LessThan by presburger
    have (∑ p ∈ ?all_indices. ?x_vec p * ?y_vec p)^2 ≤
      (∑ p ∈ ?all_indices. (?x_vec p)^2) * (∑ p ∈ ?all_indices. (?y_vec p)^2)
      using Cauchy_Schwarz_ineq_sum by blast
    with rewrite_sum_xy rewrite_sum_x2 rewrite_sum_y2 show ?thesis by simp
  qed
  have ?sum_xy ≤ sqrt (?sum_x2 * ?sum_y2)
    using real_le_sqrt cs_ineq by blast
  also have sqrt (?sum_x2 * ?sum_y2) = sqrt ?sum_x2 * sqrt ?sum_y2
    by (rule real_sqrt_mult)
  finally have 2 * ?sum_xy ≤ 2 * sqrt ?sum_x2 * sqrt ?sum_y2
    using mult_right_mono by argo
  with step1 show ?thesis by simp
qed
  have norm_square_ineq: (∑ i ∈ {0..<CARD('nr)}. ∑ j ∈ {0..<CARD('nc)}. (norm (fixed_mat_index (x + y) i j))^2) ≤

```

$$\left(\sum_{i \in \{0..<CARD('nr)\}}. \sum_{j \in \{0..<CARD('nc)\}}. (norm (fixed_mat_index\ x\ i\ j))^2\right)^2 + 2 * sqrt\left(\sum_{i \in \{0..<CARD('nr)\}}. \sum_{j \in \{0..<CARD('nc)\}}. (norm (fixed_mat_index\ x\ i\ j))^2\right) * sqrt\left(\sum_{i \in \{0..<CARD('nr)\}}. \sum_{j \in \{0..<CARD('nc)\}}. (norm (fixed_mat_index\ y\ i\ j))^2\right) + \left(\sum_{i \in \{0..<CARD('nr)\}}. \sum_{j \in \{0..<CARD('nc)\}}. (norm (fixed_mat_index\ y\ i\ j))^2\right)$$
using *sum_ineq sum_distribute cs by argo*

have *norm_square_ineq2*: $\left(\sum_{i \in \{0..<CARD('nr)\}}. \sum_{j \in \{0..<CARD('nc)\}}. (norm (fixed_mat_index\ (x + y)\ i\ j))^2\right) \leq (norm\ x)^2 + 2 * (norm\ x) * (norm\ y) + (norm\ y)^2$

proof –

have *do*: $\left(sqrt\left(\sum_{i = 0..<CARD('nr)\}. \sum_{j = 0..<CARD('nc)\}. (norm (fixed_mat_index\ x\ i\ j))^2\right)\right)^2 = \left(\sum_{i = 0..<CARD('nr)\}. \sum_{j = 0..<CARD('nc)\}. (norm (fixed_mat_index\ x\ i\ j))^2\right)$

by (*meson real_sqrt_pow2 sum_nonneg zero_le_power2*)

have *d1*: $\left(sqrt\left(\sum_{i = 0..<CARD('nr)\}. \sum_{j = 0..<CARD('nc)\}. (norm (fixed_mat_index\ y\ i\ j))^2\right)\right)^2 = \left(\sum_{i = 0..<CARD('nr)\}. \sum_{j = 0..<CARD('nc)\}. (norm (fixed_mat_index\ y\ i\ j))^2\right)$

by (*meson real_sqrt_pow2 sum_nonneg zero_le_power2*)

have $\left(\sum_{i \in \{0..<CARD('nr)\}}. \sum_{j \in \{0..<CARD('nc)\}}. (norm (fixed_mat_index\ (x + y)\ i\ j))^2\right) \leq \left(\sum_{i \in \{0..<CARD('nr)\}}. \sum_{j \in \{0..<CARD('nc)\}}. (norm (fixed_mat_index\ x\ i\ j))^2\right) + 2 * sqrt\left(\sum_{i \in \{0..<CARD('nr)\}}. \sum_{j \in \{0..<CARD('nc)\}}. (norm (fixed_mat_index\ x\ i\ j))^2\right) * sqrt\left(\sum_{i \in \{0..<CARD('nr)\}}. \sum_{j \in \{0..<CARD('nc)\}}. (norm (fixed_mat_index\ y\ i\ j))^2\right) + \left(\sum_{i \in \{0..<CARD('nr)\}}. \sum_{j \in \{0..<CARD('nc)\}}. (norm (fixed_mat_index\ y\ i\ j))^2\right)$

by (*rule norm_square_ineq*)

also have ... = $(norm\ x)^2 + 2 * (norm\ x) * (norm\ y) + (norm\ y)^2$

apply(*subst norm_def[of x]*) +

apply(*subst do*)

apply(*subst norm_def[of y]*) +

apply(*subst d1*)

using *power2_eq_square do*

by *blast*

finally show ?thesis .

qed

have $(norm\ x)^2 + 2 * (norm\ x) * (norm\ y) + (norm\ y)^2 = (norm\ x + norm\ y)^2$

by (*simp add: algebra_simps power2_sum*)

have $\left(\sum_{i \in \{0..<CARD('nr)\}}. \sum_{j \in \{0..<CARD('nc)\}}. (norm (fixed_mat_index\ (x + y)\ i\ j))^2\right) \leq (norm\ x + norm\ y)^2$

using *norm_square_ineq2* $\langle (norm\ x)^2 + 2 * (norm\ x) * (norm\ y) + (norm\ y)^2 = (norm\ x + norm\ y)^2 \rangle$ **by** *simp*

have $sqrt\left(\sum_{i \in \{0..<CARD('nr)\}}. \sum_{j \in \{0..<CARD('nc)\}}. (norm (fixed_mat_index\ (x + y)\ i\ j))^2\right) \leq sqrt\left((norm\ x + norm\ y)^2\right)$

using $\langle \left(\sum_{i \in \{0..<CARD('nr)\}}. \sum_{j \in \{0..<CARD('nc)\}}. (norm (fixed_mat_index\ (x + y)\ i\ j))^2\right) \leq (norm\ x + norm\ y)^2 \rangle$

using *real_sqrt_le_mono* **by** *blast*

have $sqrt\left((norm\ x + norm\ y)^2\right) = abs (norm\ x + norm\ y)$

by (*simp add: power2_eq_square*)

have $sqrt\left((norm\ x + norm\ y)^2\right) = norm\ x + norm\ y$

proof –

have $norm\ x \geq 0$

unfolding *norm_def*

by (*meson norm_ge_zero real_sqrt_ge_zero sum_nonneg zero_le_power*)

moreover have $norm\ y \geq 0$

unfolding *norm_def*

by (*meson norm_ge_zero real_sqrt_ge_zero sum_nonneg zero_le_power*)

ultimately have $norm\ x + norm\ y \geq 0$ **by** *simp*

hence $abs (norm\ x + norm\ y) = norm\ x + norm\ y$ **by** *simp*

thus ?thesis **using** $\langle sqrt\left((norm\ x + norm\ y)^2\right) = abs (norm\ x + norm\ y) \rangle$ **by** *simp*

qed

have $sqrt\left(\sum_{i \in \{0..<CARD('nr)\}}. \sum_{j \in \{0..<CARD('nc)\}}. (norm (fixed_mat_index\ (x + y)\ i\ j))^2\right) \leq norm\ x + norm\ y$

y

```

using <sqrt( $\sum_{i \in \{0..<CARD('nr)\}$ }.  $\sum_{j \in \{0..<CARD('nc)\}$ }. (norm (fixed_mat_index (x + y) i j))^2) ≤ sqrt((norm x + norm y)^2)>
<sqrt((norm x + norm y)^2) = norm x + norm y> by simp
thus norm (x + y) ≤ norm x + norm y
using norm_def by simp
qed

```

lemma norm_scaleR: norm (a *_R x) = |a| * norm (x::('a, 'nr::finite, 'nc::finite) fixed_mat)

proof –

```

have a0: norm (a *R x) = norm (scaleR a x)
by (simp add: scaleR_fixed_mat_def)
have a1: ... = norm (fixed_mat_smult (of_real a) x)
by (simp add: scaleR_fixed_mat_def)
have a2: ... = sqrt ( $\sum_{i \in \{0..<CARD('nr)\}$ }.  $\sum_{j \in \{0..<CARD('nc)\}$ }. (norm (fixed_mat_index (fixed_mat_smult (of_real a) x) i j))^2)
by (simp add: norm_fixed_mat_def)
have a3: ... = sqrt ( $\sum_{i \in \{0..<CARD('nr)\}$ }.  $\sum_{j \in \{0..<CARD('nc)\}$ }. (norm ((of_real a) * (fixed_mat_index x i j))^2)

```

proof –

```

have element_equality:  $\bigwedge i j. i \in \{0..<CARD('nr)\} \implies j \in \{0..<CARD('nc)\} \implies$ 
  map_fun Rep_fixed_mat id (λA i j. A $$$ (i, j)) (fixed_mat_smult (of_real a) x) i j =
  of_real a * map_fun Rep_fixed_mat id (λA i j. A $$$ (i, j)) x i j

```

proof –

```

fix i j
assume i_range: i ∈ {0..<CARD('nr)}
assume j_range: j ∈ {0..<CARD('nc)}
have b0: map_fun Rep_fixed_mat id (λA i j. A $$$ (i, j)) (fixed_mat_smult (of_real a) x) i j =
  (λA i j. A $$$ (i, j)) (Rep_fixed_mat (fixed_mat_smult (of_real a) x)) i j
by (simp add: map_fun_def)
have b1: ... = Rep_fixed_mat (fixed_mat_smult (of_real a) x) $$$ (i, j)
by simp
have b2: ... = fixed_mat_index (fixed_mat_smult (of_real a) x) i j
using i_range j_range by (simp add: fixed_mat_index_def)
have b3: ... = of_real a * fixed_mat_index x i j
using i_range j_range Rep_fixed_mat
unfolding fixed_mat_smult_def smult_mat_def fixed_mat_index_def map_mat_def

```

proof –

```

have f1:  $\forall a f. \text{Rep\_fixed\_mat (fixed\_mat\_smult a (f::('a, 'nr, 'nc) fixed\_mat))} = a \cdot_m \text{Rep\_fixed\_mat f}$ 
using fixed_mat_smult.rep_eq by blast
have f2: j < card (UNIV::'nc set)
using atLeastLessThan_iff j_range by blast
have f3: i < card (UNIV::'nr set)
using atLeastLessThan_iff i_range by blast
have f4:  $\forall f a. \text{Abs\_fixed\_mat (a} \cdot_m \text{Rep\_fixed\_mat (f::('a, 'nr, 'nc) fixed\_mat))} = \text{fixed\_mat\_smult a f}$ 
using f1 by (metis (no_types) Rep_fixed_mat_inverse)
have f5:  $\forall f. \text{dim\_col (Rep\_fixed\_mat (o::('a, 'nr, 'nc) fixed\_mat))} = \text{dim\_col (Rep\_fixed\_mat (f::('a, 'nr, 'nc) fixed\_mat))}$ 
using f1 by (smt (z3) NN_Lipschitz_Continuous.scaleR_fixed_mat_def index_smult_mat(3) scaleR_o)
then have f6: dim_col (Rep_fixed_mat (o::('a, 'nr, 'nc) fixed_mat)) = card (UNIV::'nc set)
by (smt (z3) Rep_fixed_mat_zero dim_col_mat(1) zero_mat_def)
have  $\forall f. \text{card (UNIV::'nr set)} = \text{dim\_row (Rep\_fixed\_mat (f::('a, 'nr, 'nc) fixed\_mat))}$ 
using f1 by (smt (z3) NN_Lipschitz_Continuous.scaleR_fixed_mat_def Rep_fixed_mat_zero dim_row_mat(1) index_smult_mat(2) scaleR_o zero_mat_def)
then show map_fun Rep_fixed_mat id (λm n na. m $$$ (n, na)) (map_fun id (map_fun Rep_fixed_mat Abs_fixed_mat)
(λa m. Matrix.mat (dim_row m) (dim_col m) (λp. a * m $$$ p)) (of_real a) x::('a, 'nr, 'nc) fixed_mat) i j = of_real a *

```

```

map_fun Rep_fixed_mat id (λm n na. m $$$ (n, na)) x i j
  using f6 f5 f4 f3 f2 f1 by (simp add: map_mat_def smult_mat_def)
qed
have b4: ... = of_real a * Rep_fixed_mat x $$$ (i, j)
  using i_range j_range by (simp add: fixed_mat_index_def)
have b5: ... = of_real a * (λA i j. A $$$ (i, j)) (Rep_fixed_mat x) i j
  by simp
have b6: ... = of_real a * map_fun Rep_fixed_mat id (λA i j. A $$$ (i, j)) x i j
  by (simp add: map_fun_def)
show map_fun Rep_fixed_mat id (λA i j. A $$$ (i, j)) (fixed_mat_smult (of_real a) x) i j =
  of_real a * map_fun Rep_fixed_mat id (λA i j. A $$$ (i, j)) x i j
  using bo b1 b2 b3 b4 b5 b6 by argo
qed
have co: (∑ i = 0..<CARD('nr). ∑ j = 0..<CARD('nc). (norm (map_fun Rep_fixed_mat id (λA i j. A $$$ (i, j))
(fixed_mat_smult (of_real a) x) i j))^2) =
  (∑ i = 0..<CARD('nr). ∑ j = 0..<CARD('nc). (norm (of_real a * map_fun Rep_fixed_mat id (λA i j. A $$$ (i, j)) x i
j))^2)
proof -
  have (∀ i ∈ {0..<CARD('nr)}. ∀ j ∈ {0..<CARD('nc)}).
    (norm (map_fun Rep_fixed_mat id (λA i j. A $$$ (i, j)) (fixed_mat_smult (of_real a) x) i j))^2 =
    (norm (of_real a * map_fun Rep_fixed_mat id (λA i j. A $$$ (i, j)) x i j))^2
  using element_equality by auto
  then show ?thesis by simp
qed
show ?thesis using co
  by (simp add: fixed_mat_index_def)
qed
have a4: ... = sqrt (∑ i ∈ {0..<CARD('nr)}. ∑ j ∈ {0..<CARD('nc)}. (|of_real a| * norm (fixed_mat_index x i j))^2)
  by (simp add: of_real_def)
have a5: ... = sqrt (∑ i ∈ {0..<CARD('nr)}. ∑ j ∈ {0..<CARD('nc)}. (|a| * norm (fixed_mat_index x i j))^2)
  by simp
have a6: ... = sqrt (∑ i ∈ {0..<CARD('nr)}. ∑ j ∈ {0..<CARD('nc)}. (|a|)^2 * (norm (fixed_mat_index x i j))^2)
  by (simp add: power_mult_distrib)
have a7: ... = sqrt ((|a|)^2 * (∑ i ∈ {0..<CARD('nr)}. ∑ j ∈ {0..<CARD('nc)}. (norm (fixed_mat_index x i j))^2))
  by (simp add: sum_distrib_left)
have a8: ... = |a| * sqrt (∑ i ∈ {0..<CARD('nr)}. ∑ j ∈ {0..<CARD('nc)}. (norm (fixed_mat_index x i j))^2)
  by (simp add: real_sqrt_mult)
have a9: ... = |a| * norm x
  by (simp add: norm_fixed_mat_def)
have a10: ... = |a| * norm x
  by simp
show ?thesis using a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10
  by linarith
qed

instance
  apply standard
  subgoal
    using dist_fixed_mat_def by blast
  subgoal
    unfolding plus_fixed_mat_def fixed_mat_add_def
    using Rep_fixed_mat Rep_fixed_mat_inject
    by (smt (verit, del_insts) Abs_fixed_mat_inverse add_carrier_mat assoc_add_mat)
  subgoal

```

```

unfolding plus_fixed_mat_def fixed_mat_add_def
by (metis Rep_fixed_mat comm_add_mat)
subgoal
unfolding zero_fixed_mat_def fixed_mat_zero_def plus_fixed_mat_def fixed_mat_add_def
using Rep_fixed_mat
by (metis Abs_fixed_mat_inverse Rep_fixed_mat_inverse left_add_zero_mat zero_carrier_mat)
subgoal using uminus_add by blast
subgoal unfolding minus_fixed_mat_def uminus_fixed_mat_def plus_fixed_mat_def
by simp
subgoal unfolding scaleR_fixed_mat_def fixed_mat_smult_def smult_mat_def using Rep_fixed_mat
by (smt (z3) Rep_fixed_mat_add Rep_fixed_mat_inverse add_smult_distrib_left_mat
  fixed_mat_smult.rep_eq map_fun_apply plus_fixed_mat_def smult_mat_def)
subgoal unfolding scaleR_fixed_mat_def fixed_mat_smult_def smult_mat_def using Rep_fixed_mat
by (smt (z3) Rep_fixed_mat_add Rep_fixed_mat_inverse add_smult_distrib_right_mat eq_id_iff
  fixed_mat_smult.rep_eq map_fun_apply of_real_hom.hom_add plus_fixed_mat_def smult_mat_def)
subgoal using smult by blast
subgoal using scaleR by blast
subgoal using sgn by blast
subgoal unfolding uniformity_fixed_mat_def by simp
subgoal unfolding open_fixed_mat_def by simp
subgoal using norm_eq_zero_iff by blast
subgoal using triangle_inequality by blast
subgoal using norm_scaleR by blast
done
end

```

```

instantiation fixed_vec :: (real_normed_vector, times, one, real_algebra_1}, finite) real_normed_vector
begin

```

```

lift_definition zero_fixed_vec :: ('a, 'b) fixed_vec is
zero_vec (CARD('b))
by (simp add: carrier_vec1)

```

```

lift_definition plus_fixed_vec :: ('a, 'b) fixed_vec  $\Rightarrow$  ('a, 'b) fixed_vec  $\Rightarrow$  ('a, 'b) fixed_vec is
fixed_vec_add .

```

```

definition scaleR_fixed_vec :: real  $\Rightarrow$  ('a, 'b) fixed_vec  $\Rightarrow$  ('a, 'b) fixed_vec where
scaleR_fixed_vec r A = fixed_vec_smult (of_real r) A

```

```

lift_definition uminus_fixed_vec :: ('a, 'b) fixed_vec  $\Rightarrow$  ('a, 'b) fixed_vec is
 $\lambda v. smult\_vec (-1) v$ 
unfolding smult_vec_def by auto

```

```

lift_definition minus_fixed_vec :: ('a, 'b) fixed_vec  $\Rightarrow$  ('a, 'b) fixed_vec  $\Rightarrow$  ('a, 'b) fixed_vec is
 $\lambda v w. v + (smult\_vec (-1) w)$ 
by auto

```

```

definition norm_fixed_vec :: ('a, 'b::finite) fixed_vec  $\Rightarrow$  real where
norm_fixed_vec A = sqrt ( $\sum_{i \in \{0..<CARD('b)\}}$  . (norm (fixed_vec_index A i))^2)

```

```

definition sgn_fixed_vec :: ('a, 'b::finite) fixed_vec  $\Rightarrow$  ('a, 'b) fixed_vec where
sgn_fixed_vec v = (if v = 0 then 0 else scaleR (1 / norm v) v)

```

```

definition dist_fixed_vec :: ('a, 'b) fixed_vec  $\Rightarrow$  ('a, 'b) fixed_vec  $\Rightarrow$  real where

```

$dist_fixed_vec\ v\ w = norm\ (v - w)$

definition *uniformity_fixed_vec* :: (*'a*, *'b*) *fixed_vec* × (*'a*, *'b*) *fixed_vec* *filter*
where *uniformity_fixed_vec* = (*INF* $e \in \{0 < ..\}$. *principal* $\{(x, y). dist\ x\ y < e\}$)

definition *open_fixed_vec* :: (*'a*, *'b*) *fixed_vec* *set* \Rightarrow *bool* **where**
open_fixed_vec $U = (\forall x \in U. \forall_F (x', y)$ *in* *uniformity*. $x' = x \longrightarrow y \in U)$

lemma *uminus_add_vec*: $- (A :: ('a, 'n)::finite)\ fixed_vec + A = 0$

proof –

have *a0*: *Rep_fixed_vec* $A \in carrier_vec\ CARD('n)$

using *Rep_fixed_vec* **by** *blast*

have *a1*: $-(Rep_fixed_vec\ A) + (Rep_fixed_vec\ A) = 0_v\ CARD('n)$

using *uminus_l_inv_vec* [of *Rep_fixed_vec* $A\ CARD('n)$] *a0* **by** *simp*

have *a2*: $-A + A = Abs_fixed_vec\ (Rep_fixed_vec\ (-A + A))$

by (*simp* *add*: *Rep_fixed_vec_inverse*)

have *a3*: $Rep_fixed_vec\ (-A + A) = Rep_fixed_vec\ (-A) + Rep_fixed_vec\ A$

using *Rep_fixed_vec_add* [of $-A\ A$] **unfolding** *fixed_vec_add_def*

by (*simp* *add*: *NN_Lipschitz_Continuous.plus_fixed_vec.rep_eq*)

have *a4*: $Rep_fixed_vec\ (-A) = -\ Rep_fixed_vec\ A$

using *fixed_vec_smult.rep_eq* [of $-1\ A$]

unfolding *uminus_fixed_vec_def* *fixed_vec_smult_def* *smult_vec_def*

by (*simp* *add*: *Matrix.uminus_vec_def* *NN_Lipschitz_Continuous.uminus_fixed_vec.rep_eq* *smult_vec_def*)

have *a5*: $Abs_fixed_vec\ (0_v\ CARD('n)) = Abs_fixed_vec\ (Rep_fixed_vec\ (fixed_vec_zero::('a, 'n)\ fixed_vec))$

using *Rep_fixed_vec_zero* **unfolding** *fixed_vec_zero_def*

by *metis*

have *a6*: $Abs_fixed_vec\ (Rep_fixed_vec\ (fixed_vec_zero::('a, 'n)::finite)\ fixed_vec)) =$
 $(fixed_vec_zero::('a, 'n)::finite)\ fixed_vec$

using *Rep_fixed_vec_inverse* [of $(fixed_vec_zero::('a, 'n)::finite)\ fixed_vec$] **by** *simp*

have *a7*: $(fixed_vec_zero::('a, 'n)::finite)\ fixed_vec = 0$

unfolding *zero_fixed_vec_def* *fixed_vec_zero_def* *zero_vec_def*

by (*simp* *add*: *Matrix.zero_vec_def* *NN_Lipschitz_Continuous.zero_fixed_vec_def*)

show *?thesis* **using** *a0* *a1* *a2* *a3* *a4* *a5* *a6* *a7*

by (*smt* (*verit*, *best*))

qed

lemma *smult_vec*: $a *_R b *_R x = (a * b) *_R (x::('a, 'n)::finite)\ fixed_vec$

proof –

have *b8*: $a *_R b *_R x = scaleR\ a\ (scaleR\ b\ x)$

by (*simp* *add*: *scaleR_fixed_vec_def*)

also have *b7*: $... = fixed_vec_smult\ (of_real\ a)\ (fixed_vec_smult\ (of_real\ b)\ x)$

by (*simp* *add*: *NN_Lipschitz_Continuous.scaleR_fixed_vec_def*)

also have *b6*: $... = Abs_fixed_vec\ (of_real\ a \cdot_v Rep_fixed_vec\ (fixed_vec_smult\ (of_real\ b)\ x))$

by (*simp* *add*: *fixed_vec_smult.rep_eq* *fixed_vec_smult_def*)

also have *b5*: $... = Abs_fixed_vec\ (of_real\ a \cdot_v (of_real\ b \cdot_v Rep_fixed_vec\ x))$

by (*simp* *add*: *fixed_vec_smult.rep_eq*)

also have *b4*: $... = Abs_fixed_vec\ ((of_real\ a * of_real\ b) \cdot_v Rep_fixed_vec\ x)$

using *Rep_fixed_vec*

by (*simp* *add*: *smult_smult_assoc*)

also have *b1*: $... = Abs_fixed_vec\ (of_real\ (a * b) \cdot_v Rep_fixed_vec\ x)$

by *simp*

also have *b0*: $... = fixed_vec_smult\ (of_real\ (a * b))\ x$

```

  by (simp add: fixed_vec_smult.rep_eq fixed_vec_smult_def)
  also have b3: ... = scaleR(a * b) x
  by (simp add: NN_Lipschitz_Continuous.scaleR_fixed_vec_def)
  also have b2: ... = (a * b) *R x
  by (simp add: scaleR_fixed_vec_def)
  show ?thesis
  using b0 b1 b2 b3 b4 b5 b6 b7 b8
  by argo
qed

```

lemma scaleR_vec: $1 *_{\mathbb{R}} x = (x :: ('a, 'n)::finite) \text{ fixed_vec}$

proof –

```

  have a0:  $1 *_{\mathbb{R}} x = \text{scaleR } 1 x$ 
  by (simp add: scaleR_fixed_vec_def)
  have a1: ... = fixed_vec_smult (of_real 1) x
  by (metis NN_Lipschitz_Continuous.scaleR_fixed_vec_def)
  have a2: ... = fixed_vec_smult 1 x
  by simp
  have a3: ... = x

```

proof –

```

  have fixed_vec_smult 1 x = Abs_fixed_vec (smult_vec 1 (Rep_fixed_vec x))
  by (simp add: fixed_vec_smult_def)
  also have smult_vec 1 (Rep_fixed_vec x) = Rep_fixed_vec x
  by auto

```

hence $\text{Abs_fixed_vec (smult_vec 1 (Rep_fixed_vec x))} = \text{Abs_fixed_vec (Rep_fixed_vec x)}$ **by simp**

also have ... = x **by (rule Rep_fixed_vec_inverse)**

finally show fixed_vec_smult 1 x = x .

qed

```

  show  $1 *_{\mathbb{R}} x = x$ 
  using a0 a1 a2 a3 by simp

```

qed

lemma norm_o_vec: $\text{norm } (o :: ('a, 'n)::finite) \text{ fixed_vec} = 0$

proof –

```

  have a0:  $\forall i \in \{0..<\text{CARD}('n)\}$ .
    fixed_vec_index (o :: ('a, 'n) fixed_vec) i = 0
  unfolding fixed_vec_index_def
  by (simp add: NN_Lipschitz_Continuous.zero_fixed_vec.rep_eq)
  have norm (o :: ('a, 'n) fixed_vec) =
    sqrt ( $\sum i \in \{0..<\text{CARD}('n)\}$ .
      (norm (fixed_vec_index (o :: ('a, 'n) fixed_vec) i))^2)
  by (simp add: NN_Lipschitz_Continuous.norm_fixed_vec_def a0)
  also have ... = sqrt ( $\sum i \in \{0..<\text{CARD}('n)\}$ . 0^2)
  using a0 by simp
  also have ... = sqrt 0 by simp
  finally show ?thesis by simp

```

qed

lemma scaleR_o_vec: $0 *_{\mathbb{R}} x = (o :: ('a, 'n)::finite) \text{ fixed_vec}$

proof –

```

  have a0:  $0 *_{\mathbb{R}} x = \text{scaleR } 0 x$ 

```

```

by (simp add: scaleR_fixed_vec_def)
have a1: ... = fixed_vec_smult (of_real o) x
by (simp add: NN_Lipschitz_Continuous.scaleR_fixed_vec_def)
have a2: ... = fixed_vec_smult o x
by simp
have a3: ... = o
proof -
have fixed_vec_smult o x = Abs_fixed_vec (smult_vec o (Rep_fixed_vec x))
by (simp add: fixed_vec_smult_def)
also have smult_vec o (Rep_fixed_vec x) = (Rep_fixed_vec (o :: ('a, 'n)::finite) fixed_vec))
proof -
have o ·v Rep_fixed_vec x = ov (CARD('n))
proof -
have ∀ i . ((i < (dim_vec (Rep_fixed_vec x))) →
(((o ·v Rep_fixed_vec x) $ i) = o))
unfolding smult_mat_def by auto
then have o ·v Rep_fixed_vec x = Matrix.vec CARD('n) (λij. o)
using Rep_fixed_vec[of x] by auto
moreover have Matrix.vec CARD('n) (λij. o) = ov (CARD('n))
by (simp add: zero_vec_def)
ultimately show ?thesis by simp
qed
also have ... = Rep_fixed_vec (fixed_vec_zero :: ('a, 'n)::finite) fixed_vec)
apply(subst fixed_vec_zero_def)
by (metis Rep_fixed_vec_inverse Rep_fixed_vec_zero)
finally show ?thesis unfolding zero_fixed_vec_def
by (simp add: NN_Lipschitz_Continuous.zero_fixed_vec.rep_eq)
qed
hence Abs_fixed_vec (smult_vec o (Rep_fixed_vec x)) = Abs_fixed_vec (Rep_fixed_vec (o :: ('a, 'n)::finite) fixed_vec))
by simp
also have ... = (o :: ('a, 'n)::finite) fixed_vec)
using Rep_fixed_vec_inverse by auto
finally show fixed_vec_smult o x = (o :: ('a, 'n)::finite) fixed_vec) .
qed
show o *R x = o
using a0 a1 a2 a3 by simp
qed

```

```

lemma sgn_vec: sgn x = inverse (norm x) *R (x :: ('a, 'n)::finite) fixed_vec)
proof -
have sgn x = (if x = o then o else scaleR (1 / norm x) x)
by (simp add: NN_Lipschitz_Continuous.sgn_fixed_vec_def)
show sgn x = inverse (norm x) *R x
proof (cases x = o)
case True
have inverse (norm x) *R x = inverse (norm (o :: ('a, 'n)::finite) fixed_vec)) *R o
using True by simp
also have ... = inverse o *R o
using norm_o_vec
by (metis (mono_tags, lifting))
also have ... = o
using scaleR_o_vec by simp

```

```

finally have inverse (norm x) *R x = 0 .
with True show ?thesis
  by (simp add: NN_Lipschitz_Continuous.sgn_fixed_vec_def)
next
case False
have b0: sgn x = scaleR (1 / norm x) x
  using False by (simp add: NN_Lipschitz_Continuous.sgn_fixed_vec_def)
have b1: ... = scaleR (inverse (norm x)) x
  by (simp add: divide_inverse)
have b2: ... = inverse (norm x) *R x
  by (simp add: scaleR_fixed_mat_def)
show ?thesis using b0 b1 b2 by simp
qed
qed

lemma norm_eq_zero_iff_vec: (norm x = (0::real)) = (x = (0::('a, 'n)::finite) fixed_vec))
proof
assume norm x = 0
have norm_def: norm x = sqrt (∑ i∈{0..by (simp add: NN_Lipschitz_Continuous.norm_fixed_vec_def)
with ⟨norm x = 0⟩ have b0: sqrt (∑ i∈{0..by simp
then have b1: (∑ i∈{0..by simp
then have b2: ∀ i∈{0..proof —
have non_neg: ∧ i . (norm (fixed_vec_index x i))^2 ≥ 0
  by (simp add: power2_eq_square)

thus b3: ∀ i∈{0..by (smt (verit, best) b1 finite_atLeastLessThan sum_nonneg_eq_o_iff)
qed
then have b4: ∀ i∈{0..by (simp add: power2_eq_iff)
then have b5: ∀ i∈{0..by simp
then have x = 0
  proof —
have ∀ i . i < CARD('n) → fixed_vec_index x i = 0
  using ⟨∀ i∈{0..by auto
have x = Abs_fixed_vec (Rep_fixed_vec x)
  by (simp add: Rep_fixed_vec_inverse)
also have Rep_fixed_vec x = Matrix.vec CARD('n) (λ i. fixed_vec_index x i)
  using Rep_fixed_vec unfolding fixed_vec_index_def carrier_vec_def
  by force
also have ... = Matrix.vec CARD('n) (λ i. 0)
  using ⟨∀ i . i < CARD('n) → fixed_vec_index x i = 0⟩
  by (simp add: vec_eq_iff)
also have ... = 0v CARD('n)
  unfolding zero_vec_def by auto
also have Abs_fixed_vec (0v CARD('n)) = (fixed_vec_zero::('a, 'n)::finite) fixed_vec)
  unfolding fixed_vec_zero_def by simp

```

```

also have fixed_vec_zero = 0
  using zero_fixed_vec_def
  by (simp add: fixed_vec_zero_def)
show x = 0
  by (simp add: calculation NN_Lipschitz_Continuous.zero_fixed_vec_def fixed_vec_zero_def)
qed
thus x = 0 by simp
next
assume x = 0
have norm_def: norm x = sqrt (∑ i∈{0..<CARD('n)}. (norm (fixed_vec_index x i))^2)
  by (simp add: NN_Lipschitz_Continuous.norm_fixed_vec_def)
from ⟨x = 0⟩ have ∀ i∈{0..<CARD('n)}. fixed_vec_index x i = 0
  proof -
  have ∀ i . i < CARD('n) ⟶ fixed_vec_index x i = 0
    using ⟨x::('a, 'n) fixed_vec⟩ = (0::('a, 'n) fixed_vec)⟩
      fixed_vec_index.rep_eq
    by (smt (verit) NN_Lipschitz_Continuous.zero_fixed_vec.rep_eq index_zero_vec(1))
  have x = Abs_fixed_vec (Rep_fixed_vec x)
    by (simp add: Rep_fixed_vec_inverse)
  also have Rep_fixed_vec x = Matrix.vec CARD('n) (λi. fixed_vec_index x i)
    using Rep_fixed_vec unfolding fixed_vec_index_def carrier_vec_def
    by force
  also have ... = Matrix.vec CARD('n) (λ i. 0)
    using ⟨∀ i . i < CARD('n) ⟶ fixed_vec_index x i = 0⟩
    by (simp add: vec_eq_iff)
  also have ... = 0_v CARD('n)
    by auto
  also have Abs_fixed_vec (0_v CARD('n)) = (fixed_vec_zero::('a, 'n)::finite) fixed_vec
    unfolding fixed_vec_zero_def by simp
  also have fixed_vec_zero = 0
    using zero_fixed_vec_def
    by (simp add: fixed_vec_zero_def)
  show ∀ i∈{0..<CARD('n)}. fixed_vec_index x i = 0
    using ⟨∀ (i::nat) . i < CARD('n) ⟶ fixed_vec_index (x::('a, 'n) fixed_vec) i = (0::'a)⟩ atLeastoLessThan by blast
  qed
then have ∀ i∈{0..<CARD('n)}. norm (fixed_vec_index x i) = 0
  by simp
then have ∀ i∈{0..<CARD('n)}. (norm (fixed_vec_index x i))^2 = 0
  by simp
then have (∑ i∈{0..<CARD('n)}. (norm (fixed_vec_index x i))^2) = 0
  by simp
then have sqrt (∑ i∈{0..<CARD('n)}. (norm (fixed_vec_index x i))^2) = 0
  by simp
with norm_def show norm x = 0
  by simp
qed

```

lemma triangle_inequality_vec: $\text{norm} ((x::('a, 'n)::finite) \text{fixed_vec}) + y::('a, 'n)::finite) \text{fixed_vec}) \leq \text{norm } x + \text{norm } y$

proof –

```

have norm_def: norm z = sqrt (∑ i∈{0..<CARD('n)}. (norm (fixed_vec_index z i))^2)
  for z :: ('a, 'n)::finite fixed_vec
  by (simp add: NN_Lipschitz_Continuous.norm_fixed_vec_def)
have component_add: fixed_vec_index (x + y) i = fixed_vec_index x i + fixed_vec_index y i

```

```

if  $i \in \{0..<CARD('n)\}$  for  $i$ 
proof –
  have  $fixed\_vec\_index\ (x + y)\ i = Rep\_fixed\_vec\ (x + y)\ \$\ i$ 
  by (simp add: fixed_vec_index_def)
  also have  $Rep\_fixed\_vec\ (x + y)\ \$\ i = Rep\_fixed\_vec\ ((Abs\_fixed\_vec\ (Rep\_fixed\_vec\ x + Rep\_fixed\_vec\ y))::('a,$ 
' $n)::finite)\ fixed\_vec))\ \$\ (i::nat)$ 
  by (simp add: Matrix.plus_vec_def NN_Lipschitz_Continuous.plus_fixed_vec.rep_eq fixed_vec_add_def)
  have  $Rep\_fixed\_vec\ x \in carrier\_vec\ CARD('n)$ 
  by (simp add: Rep_fixed_vec)
  moreover have  $Rep\_fixed\_vec\ y \in carrier\_vec\ CARD('n)$ 
  by (simp add: Rep_fixed_vec)
  ultimately have  $ao: Rep\_fixed\_vec\ x + Rep\_fixed\_vec\ y \in carrier\_vec\ CARD('n)$ 
  by auto
  hence  $Rep\_fixed\_vec\ (Abs\_fixed\_vec\ (Rep\_fixed\_vec\ x + Rep\_fixed\_vec\ y))::('a, 'n::finite)\ fixed\_vec) = Rep\_fixed\_vec$ 
 $x + Rep\_fixed\_vec\ y$ 
  using  $Abs\_fixed\_vec\_inverse[of\ Rep\_fixed\_vec\ x + Rep\_fixed\_vec\ y]$  by blast
  hence  $Rep\_fixed\_vec\ ((Abs\_fixed\_vec\ (Rep\_fixed\_vec\ x + Rep\_fixed\_vec\ y))::('a, 'n::finite)\ fixed\_vec))\ \$\ i =$ 
 $(Rep\_fixed\_vec\ x + Rep\_fixed\_vec\ y)\ \$\ i$ 
  by metis
  also have  $\dots = Rep\_fixed\_vec\ x\ \$\ i + Rep\_fixed\_vec\ y\ \$\ i$ 
  proof –
    have  $add\_def:$ 
 $Rep\_fixed\_vec\ (fixed\_vec\_add\ x\ y) = Rep\_fixed\_vec\ x + Rep\_fixed\_vec\ y$ 
    by auto
    have  $index\_def:$ 
 $(Rep\_fixed\_vec\ x + Rep\_fixed\_vec\ y)\ \$\ i = (Rep\_fixed\_vec\ x)\ \$\ i + (Rep\_fixed\_vec\ y)\ \$\ i$ 
    proof –
      have  $carrier\_x: Rep\_fixed\_vec\ x \in carrier\_vec\ CARD('n)$ 
      by (simp add: Rep_fixed_vec)
      have  $carrier\_y: Rep\_fixed\_vec\ y \in carrier\_vec\ CARD('n)$ 
      by (simp add: Rep_fixed_vec)
      have  $add\_def: Rep\_fixed\_vec\ x + Rep\_fixed\_vec\ y =$ 
 $vec\ CARD('n)\ (\lambda\ i'.\ Rep\_fixed\_vec\ x\ \$\ i' + Rep\_fixed\_vec\ y\ \$\ i')$ 
      using  $carrier\_x\ carrier\_y\ unfolding\ plus\_vec\_def\ vec\_def\ carrier\_vec\_def$ 
      by (simp add: cond_case_prod_eta)
      have  $(Rep\_fixed\_vec\ x + Rep\_fixed\_vec\ y)\ \$\ i =$ 
 $vec\ CARD('n)\ (\lambda\ i'.\ Rep\_fixed\_vec\ x\ \$\ i' + Rep\_fixed\_vec\ y\ \$\ i')\ \$\ i$ 
      by (simp add: add_def)
      moreover have  $vec\ CARD('n)\ (\lambda\ i'.\ Rep\_fixed\_vec\ x\ \$\ i' + Rep\_fixed\_vec\ y\ \$\ i')\ \$\ i =$ 
 $(\lambda\ i'.\ Rep\_fixed\_vec\ x\ \$\ i' + Rep\_fixed\_vec\ y\ \$\ i')\ i$ 
      using  $Rep\_fixed\_vec[of\ x]\ Rep\_fixed\_vec[of\ y]\ unfolding\ vec\_index\_def\ carrier\_vec\_def$ 
      by (metis (no_types, lifting) atLeastLessThan_iff index_vec(1) vec_index_def that(1))
      moreover have  $(\lambda\ i'.\ Rep\_fixed\_vec\ x\ \$\ i' + Rep\_fixed\_vec\ y\ \$\ i')\ i =$ 
 $Rep\_fixed\_vec\ x\ \$\ i + Rep\_fixed\_vec\ y\ \$\ i$ 
      by simp
      show ?thesis
      by (simp add: calculation(2) local.add_def)
    qed
  show ?thesis
  using  $add\_def\ index\_def$ 
  by simp
qed
also have  $\dots = fixed\_vec\_index\ x\ i + fixed\_vec\_index\ y\ i$ 
by (simp add: fixed_vec_index_def)

```

finally show $\text{fixed_vec_index } (x + y) i = \text{fixed_vec_index } x i + \text{fixed_vec_index } y i$
apply (simp add: fixed_vec_add_def fixed_vec_index.rep_eq plus_fixed_vec_def)
using <Rep_fixed_vec (x + y) \$ i = Rep_fixed_vec (Abs_fixed_vec (Rep_fixed_vec x + Rep_fixed_vec y)) \$ i> **by argo**
qed

have $\text{norm } (\text{fixed_vec_index } (x + y) i) \leq \text{norm } (\text{fixed_vec_index } x i) + \text{norm } (\text{fixed_vec_index } y i)$
if $i \in \{0..<\text{CARD } ('n)\}$ **for** i
using component_add norm_triangle_ineq **that by auto**

then have component_ineq: $(\text{norm } (\text{fixed_vec_index } (x + y) i))^2 \leq$
 $(\text{norm } (\text{fixed_vec_index } x i) + \text{norm } (\text{fixed_vec_index } y i))^2$
if $i \in \{0..<\text{CARD } ('n)\}$ **for** i
using that by simp

have expand_square: $(a + b)^2 = a^2 + 2*a*b + b^2$ **for** $a b :: \text{real}$
by (simp add: power2_eq_square algebra_simps)

have component_ineq2: $(\text{norm } (\text{fixed_vec_index } (x + y) i))^2 \leq$
 $(\text{norm } (\text{fixed_vec_index } x i))^2 + 2*(\text{norm } (\text{fixed_vec_index } x i))*(\text{norm } (\text{fixed_vec_index } y i)) +$
 $(\text{norm } (\text{fixed_vec_index } y i))^2$
if $i \in \{0..<\text{CARD } ('n)\}$ **for** i
using component_ineq expand_square **that by simp**

have sum_ineq: $(\sum i \in \{0..<\text{CARD } ('n)\}. (\text{norm } (\text{fixed_vec_index } (x + y) i))^2) \leq$
 $(\sum i \in \{0..<\text{CARD } ('n)\}. ((\text{norm } (\text{fixed_vec_index } x i))^2 +$
 $2*(\text{norm } (\text{fixed_vec_index } x i))*(\text{norm } (\text{fixed_vec_index } y i)) +$
 $(\text{norm } (\text{fixed_vec_index } y i))^2)$

proof –
have $\forall i \in \{0..<\text{CARD } ('n)\}.$
 $(\text{norm } (\text{fixed_vec_index } (x + y) i))^2 \leq$
 $(\text{norm } (\text{fixed_vec_index } x i))^2 + 2*(\text{norm } (\text{fixed_vec_index } x i))*(\text{norm } (\text{fixed_vec_index } y i)) +$
 $(\text{norm } (\text{fixed_vec_index } y i))^2$
using component_ineq2 **by blast**

have inner_sum_mono: $\forall i \in \{0..<\text{CARD } ('n)\}.$
 $((\text{norm } (\text{fixed_vec_index } (x + y) i))^2) \leq$
 $((\text{norm } (\text{fixed_vec_index } x i))^2 +$
 $2 * \text{norm } (\text{fixed_vec_index } x i) * \text{norm } (\text{fixed_vec_index } y i) +$
 $(\text{norm } (\text{fixed_vec_index } y i))^2)$
using component_ineq2 **by blast**

have $(\sum i \in \{0..<\text{CARD } ('n)\}. (\text{norm } (\text{fixed_vec_index } (x + y) i))^2) \leq$
 $(\sum i \in \{0..<\text{CARD } ('n)\}. (\text{norm } (\text{fixed_vec_index } x i))^2 +$
 $2 * \text{norm } (\text{fixed_vec_index } x i) * \text{norm } (\text{fixed_vec_index } y i) +$
 $(\text{norm } (\text{fixed_vec_index } y i))^2)$
using inner_sum_mono sum_mono **by fast**

thus ?thesis .

qed

have sum_distribute: $(\sum i \in \{0..<\text{CARD } ('n)\}. (\text{norm } (\text{fixed_vec_index } x i))^2 +$
 $2*(\text{norm } (\text{fixed_vec_index } x i))*(\text{norm } (\text{fixed_vec_index } y i)) +$
 $(\text{norm } (\text{fixed_vec_index } y i))^2) =$
 $(\sum i \in \{0..<\text{CARD } ('n)\}. (\text{norm } (\text{fixed_vec_index } x i))^2) +$
 $(\sum i \in \{0..<\text{CARD } ('n)\}. 2*(\text{norm } (\text{fixed_vec_index } x i))*(\text{norm } (\text{fixed_vec_index } y i))) +$
 $(\sum i \in \{0..<\text{CARD } ('n)\}. (\text{norm } (\text{fixed_vec_index } y i))^2)$
by (simp add: sum.distrib)

have cs: $(\sum i \in \{0..<\text{CARD } ('n)\}. 2*(\text{norm } (\text{fixed_vec_index } x i))*(\text{norm } (\text{fixed_vec_index } y i))) \leq$
 $2 * \text{sqrt}(\sum i \in \{0..<\text{CARD } ('n)\}. (\text{norm } (\text{fixed_vec_index } x i))^2) *$
 $\text{sqrt}(\sum i \in \{0..<\text{CARD } ('n)\}. (\text{norm } (\text{fixed_vec_index } y i))^2)$

proof –
let ?x_norm_i = $\lambda i . \text{norm } (\text{fixed_vec_index } x i)$

```

let ?y_norm_i = λi . norm (fixed_vec_index y i)
let ?sum_xy = ∑ i = 0..<CARD('n). ?x_norm_i i * ?y_norm_i i
let ?sum_x2 = ∑ i = 0..<CARD('n). (?x_norm_i i)^2
let ?sum_y2 = ∑ i = 0..<CARD('n). (?y_norm_i i)^2
have step1: (∑ i = 0..<CARD('n). 2 * ?x_norm_i i * ?y_norm_i i) =
  2 * (∑ i = 0..<CARD('n). ?x_norm_i i * ?y_norm_i i)
using sum_distrib_left[of 2 _ {0..<CARD('nr)}]
proof -
let ?g = λi . norm (fixed_vec_index x i) * norm (fixed_vec_index y i)
have 2 * (∑ i = 0..<CARD('n). ?g i) = (∑ i = 0..<CARD('n). 2 * ?g i)
  using sum_distrib_left[of 2 ?g {0..<CARD('nr)}] by blast
have (∑ i = 0..<CARD('n). 2 * (norm (fixed_vec_index x i) * norm (fixed_vec_index y i))) =
  (∑ i = 0..<CARD('n). (2 * norm (fixed_vec_index x i) * norm (fixed_vec_index y i)))
  by (meson vector_space_over_itself.vector_space_assms(3))

show ?thesis using <2 * (∑ i = 0..<CARD('n). ?g i) = (∑ i = 0..<CARD('n). 2 * ?g i)>
  and <(∑ i = 0..<CARD('n). 2 * (norm (fixed_vec_index x i) * norm (fixed_vec_index y i))) =
  (∑ i = 0..<CARD('n). (2 * norm (fixed_vec_index x i) * norm (fixed_vec_index y i)))>
  by simp
qed
have cs_ineq: ?sum_xy^2 ≤ ?sum_x2 * ?sum_y2
  by (simp add: Cauchy_Schwarz_ineq_sum)
have ?sum_xy ≤ sqrt (?sum_x2 * ?sum_y2)
  using real_le_sqrt cs_ineq by blast
also have sqrt (?sum_x2 * ?sum_y2) = sqrt ?sum_x2 * sqrt ?sum_y2
  by (rule real_sqrt_mult)
finally have 2 * ?sum_xy ≤ 2 * sqrt ?sum_x2 * sqrt ?sum_y2
  using mult_right_mono by argo
with step1 show ?thesis by simp
qed
have norm_square_ineq: (∑ i ∈ {0..<CARD('n)}. (norm (fixed_vec_index (x + y) i))^2) ≤
  (∑ i ∈ {0..<CARD('n)}. (norm (fixed_vec_index x i))^2) +
  2 * sqrt(∑ i ∈ {0..<CARD('n)}. (norm (fixed_vec_index x i))^2) *
  sqrt(∑ i ∈ {0..<CARD('n)}. (norm (fixed_vec_index y i))^2) +
  (∑ i ∈ {0..<CARD('n)}. (norm (fixed_vec_index y i))^2)
using sum_ineq sum_distribute cs by argo
have norm_square_ineq2: (∑ i ∈ {0..<CARD('n)}. (norm (fixed_vec_index (x + y) i))^2) ≤
  (norm x)^2 + 2 * (norm x) * (norm y) + (norm y)^2
proof -
have do: (sqrt (∑ i = 0..<CARD('n). (norm (fixed_vec_index x i))^2))^2 =
  (∑ i = 0..<CARD('n). (norm (fixed_vec_index x i))^2)
  by (meson real_sqrt_pow2 sum_nonneg_zero_le_power2)
have d1: (sqrt (∑ i = 0..<CARD('n). (norm (fixed_vec_index y i))^2))^2 =
  (∑ i = 0..<CARD('n). (norm (fixed_vec_index y i))^2)
  by (meson real_sqrt_pow2 sum_nonneg_zero_le_power2)
have (∑ i ∈ {0..<CARD('n)}. (norm (fixed_vec_index (x + y) i))^2) ≤
  (∑ i ∈ {0..<CARD('n)}. (norm (fixed_vec_index x i))^2) +
  2 * sqrt(∑ i ∈ {0..<CARD('n)}. (norm (fixed_vec_index x i))^2) *
  sqrt(∑ i ∈ {0..<CARD('n)}. (norm (fixed_vec_index y i))^2) +
  (∑ i ∈ {0..<CARD('n)}. (norm (fixed_vec_index y i))^2)
  by (rule norm_square_ineq)
also have ... = (norm x)^2 + 2 * (norm x) * (norm y) + (norm y)^2
  apply(subst norm_def[of x]) +
  apply(subst do)

```

```

    apply(subst norm_def[of y])+
    apply(subst d1)
    using power2_eq_square do
    by blast
  finally show ?thesis .
qed
have (norm x)^2 + 2 * (norm x) * (norm y) + (norm y)^2 = (norm x + norm y)^2
  by (simp add: algebra_simps power2_sum)
have  $\sum_{i \in \{0..<CARD('n)\}}. (norm (fixed_vec_index (x + y) i))^2 \leq (norm x + norm y)^2$ 
  using norm_square_ineq2 <(norm x)^2 + 2 * (norm x) * (norm y) + (norm y)^2 = (norm x + norm y)^2> by simp
have sqrt( $\sum_{i \in \{0..<CARD('n)\}}. (norm (fixed_vec_index (x + y) i))^2 \leq \sqrt{(norm x + norm y)^2}$ )
  using <(math>\sum_{i \in \{0..<CARD('n)\}}. (norm (fixed_vec_index (x + y) i))^2 \leq (norm x + norm y)^2>
  using real_sqrt_le_mono by blast
have sqrt((norm x + norm y)^2) = abs (norm x + norm y)
  by (simp add: power2_eq_square)
have sqrt((norm x + norm y)^2) = norm x + norm y
proof -
  have norm x ≥ 0
    unfolding norm_def
    by (meson norm_ge_zero real_sqrt_ge_zero sum_nonneg zero_le_power)
  moreover have norm y ≥ 0
    unfolding norm_def
    by (meson norm_ge_zero real_sqrt_ge_zero sum_nonneg zero_le_power)
  ultimately have norm x + norm y ≥ 0 by simp
  hence abs (norm x + norm y) = norm x + norm y by simp
  thus ?thesis using <sqrt((norm x + norm y)^2) = abs (norm x + norm y)> by simp
qed
have sqrt( $\sum_{i \in \{0..<CARD('n)\}}. (norm (fixed_vec_index (x + y) i))^2 \leq norm x + norm y$ )
  using <sqrt( $\sum_{i \in \{0..<CARD('n)\}}. (norm (fixed_vec_index (x + y) i))^2 \leq \sqrt{(norm x + norm y)^2}$ >
  <sqrt((norm x + norm y)^2) = norm x + norm y> by simp
  thus norm (x + y) ≤ norm x + norm y
    using norm_def by simp
qed

```

lemma norm_scaleR_vec: $norm (a *_{\mathbb{R}} x) = |a| * norm (x::('a, 'n)::finite) fixed_vec$

proof —

```

have a0: norm (a *ℝ x) = norm (scaleR a x)
  by simp

```

```

have a1: ... = norm (fixed_vec_smult (of_real a) x)
  by (simp add: NN_Lipschitz_Continuous.scaleR_fixed_vec_def)

```

```

have a2: ... = sqrt ( $\sum_{i \in \{0..<CARD('n)\}}. (norm (fixed_vec_index (fixed_vec_smult (of_real a) x) i))^2$ )
  by (simp add: NN_Lipschitz_Continuous.norm_fixed_vec_def)

```

```

have a3: ... = sqrt ( $\sum_{i \in \{0..<CARD('n)\}}. (norm ((of_real a) * (fixed_vec_index x i)))^2$ )

```

proof —

```

have element_equality:  $\bigwedge i . i \in \{0..<CARD('n)\} \implies$ 
  map_fun Rep_fixed_vec id ( $\lambda A i . A \$ i$ ) (fixed_vec_smult (of_real a) x) i =
  of_real a * map_fun Rep_fixed_vec id ( $\lambda A i . A \$ i$ ) x i

```

proof —

fix i

assume i_range: $i \in \{0..<CARD('n)\}$

```

have b0: map_fun Rep_fixed_vec id ( $\lambda A i . A \$ i$ ) (fixed_vec_smult (of_real a) x) i =
  ( $\lambda A i . A \$ i$ ) (Rep_fixed_vec (fixed_vec_smult (of_real a) x)) i

```

```

  by (simp add: map_fun_def)
  have b1: ... = Rep_fixed_vec (fixed_vec_smult (of_real a) x) $ i
  by simp
  have b2: ... = fixed_vec_index (fixed_vec_smult (of_real a) x) i
  using i_range by (simp add: fixed_vec_index_def)
  have b3: ... = of_real a * fixed_vec_index x i
  using i_range Rep_fixed_vec
  unfolding fixed_vec_smult_def smult_vec_def fixed_vec_index_def map_vec_def
  proof -
  have f1:  $\forall a f. \text{Rep\_fixed\_vec} (\text{fixed\_vec\_smult } a (f::('a, 'n) \text{ fixed\_vec})) = a \cdot_v \text{Rep\_fixed\_vec } f$ 
  using fixed_vec_smult.rep_eq by blast
  have f3:  $i < \text{card} (\text{UNIV}::'n \text{ set})$ 
  using atLeastLessThan_iff i_range by blast
  have f4:  $\forall f a. \text{Abs\_fixed\_vec} (a \cdot_v \text{Rep\_fixed\_vec} (f::('a, 'n) \text{ fixed\_vec})) = \text{fixed\_vec\_smult } a f$ 
  using f1 by (metis (no_types) Rep_fixed_vec_inverse)
  have f5:  $\forall f. \text{dim\_vec} (\text{Rep\_fixed\_vec} (o::('a, 'n) \text{ fixed\_vec})) = \text{dim\_vec} (\text{Rep\_fixed\_vec} (f::('a, 'n) \text{ fixed\_vec}))$ 
  using f1 by (smt (z3) NN_Lipschitz_Continuous.scaleR_fixed_vec_def index_smult_vec scaleR_o_vec)
  then have f6:  $\text{dim\_vec} (\text{Rep\_fixed\_vec} (o::('a, 'n) \text{ fixed\_vec})) = \text{card} (\text{UNIV}::'n \text{ set})$ 
  by (smt (z3) Rep_fixed_vec_zero dim_vec zero_vec_def)
  have  $\forall f. \text{card} (\text{UNIV}::'n \text{ set}) = \text{dim\_vec} (\text{Rep\_fixed\_vec} (f::('a, 'n) \text{ fixed\_vec}))$ 
  using f5 f6 by force
  then show map_fun Rep_fixed_vec id ( $\lambda m n. m \$ n$ ) (map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) ( $\lambda a m. \text{Matrix.vec} (\text{dim\_vec } m) (\lambda p. a * m \$ p)$ ) (of_real a) x::('a, 'n) fixed_vec) i
  = of_real a * map_fun Rep_fixed_vec id ( $\lambda m n. m \$ n$ ) x i
  using f6 f5 f4 f3 f1 by (simp add: map_vec_def smult_vec_def)
  qed
  have b4: ... = of_real a * Rep_fixed_vec x $ i
  using i_range by (simp add: fixed_vec_index_def)
  have b5: ... = of_real a * ( $\lambda i. A \$ i$ ) (Rep_fixed_vec x) i
  by simp
  have b6: ... = of_real a * map_fun Rep_fixed_vec id ( $\lambda i. A \$ i$ ) x i
  by (simp add: map_fun_def)
  show map_fun Rep_fixed_vec id ( $\lambda i. A \$ i$ ) (fixed_vec_smult (of_real a) x) i =
  of_real a * map_fun Rep_fixed_vec id ( $\lambda i. A \$ i$ ) x i
  using b0 b1 b2 b3 b4 b5 b6 by argo
  qed
  have co:  $(\sum i = 0..<\text{CARD}('n). (\text{norm} (\text{map\_fun } \text{Rep\_fixed\_vec } \text{id} (\lambda i. A \$ i) (\text{fixed\_vec\_smult} (\text{of\_real } a) x) i))^2) =$ 
 $(\sum i = 0..<\text{CARD}('n). (\text{norm} (\text{of\_real } a * \text{map\_fun } \text{Rep\_fixed\_vec } \text{id} (\lambda i. A \$ i) x) i))^2$ 
  using element_equality by force
  show ?thesis using co
  by (simp add: fixed_vec_index_def)
  qed
  have a4: ... =  $\text{sqrt} (\sum i \in \{0..<\text{CARD}('n)\}. (|\text{of\_real } a| * \text{norm} (\text{fixed\_vec\_index } x i))^2)$ 
  by (simp add: of_real_def)
  have a5: ... =  $\text{sqrt} (\sum i \in \{0..<\text{CARD}('n)\}. (|a| * \text{norm} (\text{fixed\_vec\_index } x i))^2)$ 
  by simp
  have a6: ... =  $\text{sqrt} (\sum i \in \{0..<\text{CARD}('n)\}. (|a|)^2 * (\text{norm} (\text{fixed\_vec\_index } x i))^2)$ 
  by (simp add: power_mult_distrib)
  have a7: ... =  $\text{sqrt} ((|a|)^2 * (\sum i \in \{0..<\text{CARD}('n)\}. (\text{norm} (\text{fixed\_vec\_index } x i))^2))$ 
  by (simp add: sum_distrib_left)
  have a8: ... =  $|a| * \text{sqrt} (\sum i \in \{0..<\text{CARD}('n)\}. (\text{norm} (\text{fixed\_vec\_index } x i))^2)$ 
  by (simp add: real_sqrt_mult)
  have a9: ... =  $|a| * \text{norm } x$ 
  by (simp add: NN_Lipschitz_Continuous.norm_fixed_vec_def)

```

```

have a10: ... = |a| * norm x
  by simp
show ?thesis using a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10
  by linarith
qed

```

instance

```

apply standard
subgoal using dist_fixed_vec_def by simp
subgoal unfolding plus_fixed_vec_def fixed_vec_add_def
  using add_carrier_vec assoc_add_vec
  by (smt (verit, del_insts) Abs_fixed_vec_inverse Rep_fixed_vec carrier_vec_def id_apply)
subgoal unfolding plus_fixed_vec_def fixed_vec_add_def
  using Rep_fixed_vec comm_add_vec
  by (metis id_def)
subgoal unfolding zero_fixed_vec_def plus_fixed_vec_def fixed_vec_add_def id_def
  using Rep_fixed_vec Abs_fixed_vec_inverse Rep_fixed_vec_inverse zero_carrier_vec
  by (metis left_zero_vec)
subgoal using uminus_add_vec by blast
subgoal
  by (metis (mono_tags, lifting) NN_Lipschitz_Continuous.minus_fixed_vec.rep_eq
    NN_Lipschitz_Continuous.plus_fixed_vec.transfer Rep_fixed_vec_inverse
    fixed_vec_add_def uminus_fixed_vec.rep_eq)
subgoal
  by (smt (verit) NN_Lipschitz_Continuous.scaleR_fixed_vec_def Rep_fixed_vec
    Rep_fixed_vec_add fixed_vec.Rep_fixed_vec_inject fixed_vec_smult.rep_eq
    plus_fixed_vec.abs_eq smult_add_distrib_vec)
subgoal
  by (metis (mono_tags, opaque_lifting) NN_Lipschitz_Continuous.plus_fixed_vec.rep_eq
    Rep_fixed_vec_add add_smult_distrib_vec fixed_vec.Rep_fixed_vec_inject
    fixed_vec_smult.rep_eq of_real_hom.hom_add scaleR_fixed_vec_def)
subgoal using smult_vec by blast
subgoal using scaleR_vec by blast
subgoal using sgn_vec by blast
subgoal unfolding uniformity_fixed_vec_def by blast
subgoal unfolding open_fixed_vec_def by blast
subgoal using norm_eq_zero_iff_vec by blast
subgoal using triangle_inequality_vec by blast
subgoal using norm_scaleR_vec by blast
done

```

end

lemma uminus_fixed_vec:

```

assumes (v::'a::{real_algebra_1,real_normed_vector} Matrix.vec) ∈ carrier_vec (CARD('n::finite))
shows - Abs_fixed_vec v = (Abs_fixed_vec (- v))::('a::{real_algebra_1,real_normed_vector}, 'n::finite) fixed_vec
proof -
have Rep_fixed_vec ((Abs_fixed_vec v)::('a::{real_algebra_1,real_normed_vector}, 'n::finite) fixed_vec) = v
  using assms Abs_fixed_vec_inverse[of v]
  by blast
hence Rep_fixed_vec (- (Abs_fixed_vec v))::('a::{real_algebra_1,real_normed_vector}, 'n::finite) fixed_vec = - v
  using uminus_fixed_vec.rep_eq
proof -
have f1: Rep_fixed_vec (Abs_fixed_vec v)::('a, 'n) fixed_vec = v

```

```

    using <Rep_fixed_vec (Abs_fixed_vec (v::'a Matrix.vec)) = v> by blast
    have  $\forall f. \text{Rep\_fixed\_vec } (- (f::('a, 'n) \text{fixed\_vec})) = - 1 \cdot_v \text{Rep\_fixed\_vec } f$ 
      using uminus_fixed_vec.rep_eq by blast
    then show ?thesis
      using f1 by auto
  qed
  thus ?thesis
    by (metis Rep_fixed_vec_inverse)
qed

```

lemma lipschitz_on_mat_add:

```

  shows <(1::real)–lipschitz_on U ( $\lambda (A::('a::\{\text{real\_algebra\_1, real\_normed\_vector}\}, 'nr::\text{finite}, 'nc::\text{finite}) \text{fixed\_mat}) .$ 
  A + M)>
  by (simp add: lipschitz_on1)

```

lemma vec_minus_element:

```

  fixes v w :: 'a::\{minus, zero\} vec
  assumes dim_vec v = dim_vec w and i < dim_vec v
  shows vec_index (v - w) i = vec_index v i - vec_index w i
proof -
  from assms have dim_eq: dim_vec (v - w) = dim_vec v
    by simp
  have vec_index (v - w) i = vec_index (vec (dim_vec v) ( $\lambda j. \text{vec\_index } v \ j - \text{vec\_index } w \ j$ )) i
    using assms by simp
  also have ... = ( $\lambda j. \text{vec\_index } v \ j - \text{vec\_index } w \ j$ ) i
    using assms dim_eq by simp
  also have ... = vec_index v i - vec_index w i
    by simp
  finally show ?thesis
    by (metis assms(1) assms(2) index_minus_vec(1))
qed

```

lemma vec_minus:

```

  fixes v w :: 'a::\{minus, zero\} vec
  assumes dim_vec v = dim_vec w and i < dim_vec v
  shows (v - w) = vec (dim_vec v) ( $\lambda i. \text{vec\_index } v \ i - \text{vec\_index } w \ i$ )
proof -
  have v - w = vec (dim_vec (v - w)) ( $\lambda i. \text{vec\_index } (v - w) \ i$ )
    by (simp add: vec_eq_iff)
  also have dim_vec (v - w) = dim_vec v
    using assms vec_minus_element
    by (metis index_minus_vec(2))
  also have Matrix.vec (dim_vec v) ( $\lambda i. (v - w) \ \$ \ i$ ) = Matrix.vec (dim_vec v) ( $\lambda i. v \ \$ \ i - w \ \$ \ i$ )
    using assms(1) by fastforce
  finally show ?thesis by simp
qed

```

lemma Rep_fixed_vec_plus:

```

  Rep_fixed_vec ((u::('a::\{real\_algebra\_1, real\_normed\_vector\}, 'n::\text{finite}) \text{fixed\_vec}) +
    (v::('a::\{real\_algebra\_1, real\_normed\_vector\}, 'n::\text{finite}) \text{fixed\_vec})) =
  Rep_fixed_vec u + Rep_fixed_vec v
proof -
  have u + v  $\in \{v. \text{Rep\_fixed\_vec } v \in \text{carrier\_vec } (\text{CARD}('n))\}$ 
    using Rep_fixed_vec_unfolding carrier_vec_def

```

by blast
thus $\text{Rep_fixed_vec } (u + v) = \text{Rep_fixed_vec } u + \text{Rep_fixed_vec } v$
using $\text{Rep_fixed_vec unfolding plus_fixed_vec_def}$
by force
qed

lemma *fixed_vec_add*:

assumes $v1 \in \text{carrier_vec } (\text{CARD } ('n::\text{finite}))$
and $v2 \in \text{carrier_vec } (\text{CARD } ('n::\text{finite}))$
shows $\text{Abs_fixed_vec } v1 + \text{Abs_fixed_vec } v2 = (\text{Abs_fixed_vec } (v1 + v2))::('a::\{\text{real_algebra_1, real_normed_vector}\}, 'n) \text{ fixed_vec}$

proof –

have $\text{carrier_sum}: v1 + v2 \in \text{carrier_vec } (\text{CARD } ('n))$
using $\text{assms add_carrier_vec by blast}$
have $\text{Rep_fixed_vec } ((\text{Abs_fixed_vec } v1 + \text{Abs_fixed_vec } v2)::('a, 'n) \text{ fixed_vec}) =$
 $\text{Rep_fixed_vec } ((\text{Abs_fixed_vec } v1)::('a, 'n) \text{ fixed_vec}) +$
 $\text{Rep_fixed_vec } ((\text{Abs_fixed_vec } v2)::('a, 'n) \text{ fixed_vec})$
using $\text{Rep_fixed_vec_plus[of Abs_fixed_vec } v1 \text{ Abs_fixed_vec } v2]$
by blast
also have $\text{Rep_fixed_vec } ((\text{Abs_fixed_vec } v1)::('a, 'n) \text{ fixed_vec}) +$
 $\text{Rep_fixed_vec } ((\text{Abs_fixed_vec } v2)::('a, 'n) \text{ fixed_vec}) =$
 $v1 + v2$
using $\text{assms Abs_fixed_vec_inverse[of } v1]$
 $\text{Abs_fixed_vec_inverse[of } v2]$
by (*metis calculation*)
also have $\dots = \text{Rep_fixed_vec } ((\text{Abs_fixed_vec } (v1 + v2))::('a, 'n) \text{ fixed_vec})$
using carrier_sum assms
by (*metis Rep_fixed_vec_inverse calculation*)
finally have $\text{Rep_fixed_vec } ((\text{Abs_fixed_vec } v1 + \text{Abs_fixed_vec } v2)::('a, 'n) \text{ fixed_vec}) =$
 $\text{Rep_fixed_vec } ((\text{Abs_fixed_vec } (v1 + v2))::('a, 'n) \text{ fixed_vec}) .$
thus $\text{Abs_fixed_vec } v1 + \text{Abs_fixed_vec } v2 = (\text{Abs_fixed_vec } (v1 + v2))::('a, 'n) \text{ fixed_vec}$
using $\text{Rep_fixed_vec_inject by blast}$

qed

lemma *col_minus_mat*:

fixes $A B :: 'a::\{\text{minus, zero}\} \text{ mat}$
assumes $\text{dim_row } A = \text{dim_row } B$ **and** $\text{dim_col } A = \text{dim_col } B$ **and** $i < \text{dim_col } A$
shows $\text{col } (A - B) i = \text{col } A i - \text{col } B i$

proof –

from $\text{assms have dim_col_eq}: \text{dim_col } (A - B) = \text{dim_col } A$
by (*metis index_minus_mat(3)*)
from $\text{assms have dim_row_eq}: \text{dim_row } (A - B) = \text{dim_row } A$
by (*metis index_minus_mat(2)*)
have $\text{col } (A - B) i = \text{col } A i - \text{col } B i$
proof –
have $\text{col } (A - B) i = \text{vec } (\text{dim_row } (A - B)) (\lambda j. (A - B) \$\$ (j, i))$
by (*simp add: col_def*)
also have $\dots = \text{vec } (\text{dim_row } A) (\lambda j. (A - B) \$\$ (j, i))$
using $\text{dim_row_eq assms(1) by presburger}$
also have $\dots = \text{vec } (\text{dim_row } A) (\lambda j. A \$\$ (j, i) - B \$\$ (j, i))$
using $\text{assms } \langle (i::\text{nat}) < \text{dim_col } (A::'a \text{ mat}) \rangle \text{ by fastforce}$
also have $\dots = \text{vec } (\text{dim_row } A) (\lambda j. A \$\$ (j, i)) - \text{vec } (\text{dim_row } A) (\lambda j. B \$\$ (j, i))$
by (*simp add: Matrix.vec_eq_iff*)

also have ... = col A i - col B i
using *assms*(1) col_def **by** *metis*
finally show col (A - B) i = col A i - col B i.
qed
show col (A - B) i = col A i -

col B i
using *assms* **by** *auto*
qed

lemma *index_vec_mat_mult*:
assumes $v \in \text{carrier_vec } (\text{dim_row } A)$
and $A \in \text{carrier_mat } (\text{dim_row } A) (\text{dim_col } A)$
and $i < \text{dim_col } (A::'a::\{\text{semiring_o}, \text{ab_semigroup_mult}\} \text{Matrix.mat})$
shows $(v \cdot v * A) \$ i = (\sum j = 0..<\text{dim_row } A. v \$ j * A \$\$ (j, i))$
proof -
have *ao*: $(v \cdot v * A) \$ i = (\text{Matrix.vec } (\text{dim_col } A) (\lambda k. \text{col } A k \cdot v)) \$ i$
using *assms* **unfolding** *mult_vec_mat_def* **by** *blast*
have *a1*: $(\text{Matrix.vec } (\text{dim_col } A) (\lambda k. \text{col } A k \cdot v)) \$ i = (\lambda k. \text{col } A k \cdot v) i$
using *assms* *index_vec* **by** *blast*
have *a2*: ... = col A i · v
using *assms* **by** *blast*
have *a3*: ... = $(\sum j = 0..<\text{dim_row } A. \text{col } A i \$ j * v \$ j)$
using *assms* *scalar_prod_def*
by (*metis* *carrier_vecD*)
have *a4*: $(\sum j = 0..<\text{dim_row } A. \text{col } A i \$ j * v \$ j) = (\sum j = 0..<\text{dim_row } A. A \$\$ (j, i) * v \$ j)$
using *assms* *index_col*
by (*metis* (*no_types*, *lifting*) *sum.ivl_cong*)
have *a5*: ... = $(\sum j = 0..<\text{dim_row } A. v \$ j * A \$\$ (j, i))$
using *mult.commute* **by** *metis*
show $(v \cdot v * A) \$ i = (\sum j = 0..<\text{dim_row } A. v \$ j * A \$\$ (j, i))$
using *ao a1 a2 a3 a4 a5* **by** *argo*
qed

lemma *Rep_fixed_mat_minus*:
 $\text{Rep_fixed_mat } ((x - y)::('a, 'b, 'c) \text{fixed_mat}) = \text{Rep_fixed_mat } x - \text{Rep_fixed_mat } (y::('a::\{\text{real_algebra_1}, \text{real_normed_vector}\}, 'b::\text{finite}, 'c::\text{finite}) \text{fixed_mat})$
proof -
have $\text{Rep_fixed_mat } (x - y) = \text{Rep_fixed_mat } (\text{Abs_fixed_mat } (\text{Rep_fixed_mat } x - \text{Rep_fixed_mat } y)::('a, 'b, 'c) \text{fixed_mat})$
proof -
have $x - y = \text{Abs_fixed_mat } (\text{Rep_fixed_mat } x - \text{Rep_fixed_mat } y)$
proof -
have $x - y = \text{fixed_mat_add } x (\text{fixed_mat_smult } (-1) y::('a, 'b, 'c) \text{fixed_mat})$
using *minus_fixed_mat_def* **by** *blast*
also have ... = $\text{Abs_fixed_mat } (\text{Rep_fixed_mat } x + \text{Rep_fixed_mat } (\text{fixed_mat_smult } (-1) y::('a, 'b, 'c) \text{fixed_mat}))$
by (*simp* *add:fixed_mat_add_def*)
also have ... = $\text{Abs_fixed_mat } (\text{Rep_fixed_mat } x + (-1) \cdot_m \text{Rep_fixed_mat } (y::('a, 'b, 'c) \text{fixed_mat}))$
by (*simp* *add:fixed_mat_smult.rep_eq*)
also have ... = $\text{Abs_fixed_mat } (\text{Rep_fixed_mat } x + (- \text{Rep_fixed_mat } (y::('a, 'b, 'c) \text{fixed_mat})))$
by (*smt* (*verit*, *ccfv_SIG*) *Rep_fixed_mat* *Rep_fixed_mat_add*
 $\langle \text{Abs_fixed_mat } (\text{Rep_fixed_mat } (x::('a, 'b, 'c) \text{fixed_mat}) + \text{Rep_fixed_mat } (\text{fixed_mat_smult } (-1) y::('a, 'b, 'c) \text{fixed_mat})) = \text{Abs_fixed_mat } (\text{Rep_fixed_mat } x + (-1) \cdot_m \text{Rep_fixed_mat } y) \rangle$
 $\langle \text{fixed_mat_add } (x::('a, 'b, 'c) \text{fixed_mat}) (\text{fixed_mat_smult } (-1) y::('a, 'b, 'c) \text{fixed_mat}) = \text{Abs_fixed_mat}$

```

(Rep_fixed_mat x + Rep_fixed_mat (fixed_mat_smult (- (1::'a) y)))
  assoc_add_mat calculation comm_add_mat diff_add_cancel plus_fixed_mat_def
  right_add_zero_mat uminus_carrier_iff_mat uminus_l_inv_mat
also have ... = Abs_fixed_mat (Rep_fixed_mat x - Rep_fixed_mat y)
by (metis Rep_fixed_mat add_uminus_minus_mat)
finally show x - y = Abs_fixed_mat (Rep_fixed_mat x - Rep_fixed_mat y) .
qed
thus Rep_fixed_mat ((x - y)::('a, 'b, 'c) fixed_mat) = Rep_fixed_mat ((Abs_fixed_mat (Rep_fixed_mat x -
Rep_fixed_mat y))::('a, 'b, 'c) fixed_mat)
by argo
qed
also have Rep_fixed_mat (Abs_fixed_mat (Rep_fixed_mat x - Rep_fixed_mat y)::('a, 'b, 'c) fixed_mat) = Rep_fixed_mat
x - Rep_fixed_mat y
proof -
have Rep_fixed_mat x ∈ carrier_mat (CARD('b)) (CARD('c))
using Rep_fixed_mat[of x] by blast
have Rep_fixed_mat y ∈ carrier_mat (CARD('b)) (CARD('c))
by (rule Rep_fixed_mat)
hence Rep_fixed_mat x - Rep_fixed_mat y ∈ carrier_mat (CARD('b)) (CARD('c))
using <Rep_fixed_mat x ∈ carrier_mat (CARD('b)) (CARD('c))>
using minus_carrier_mat by blast
thus ?thesis
by (metis Abs_fixed_mat_inverse calculation)
qed
finally show Rep_fixed_mat (x - y) = Rep_fixed_mat x - Rep_fixed_mat y .
qed

```

lemma vector_matrix_inequality:

```

fixes c :: ('a::{real_normed_field,real_inner}, 'nr::finite) fixed_vec
and U :: ('a, 'nr, 'nc::finite) fixed_mat set
and C :: real
assumes C_bound: C ≥ norm c
and C_nonneg: C ≥ 0
shows  $\bigwedge x y. x \in U \implies y \in U \implies$ 
sqrt ( $\sum i = 0..<CARD('nc). (norm (fixed\_vec\_index (c_{fv} * x - c_{fv} * y) i))^2$ ) ≤
C * sqrt ( $\sum i = 0..<CARD('nr). \sum j = 0..<CARD('nc). (norm (fixed\_mat\_index (x - y) i j))^2$ )
proof -
fix x y :: ('a, 'nr, 'nc) fixed_mat
assume x_in_U: x ∈ U and y_in_U: y ∈ U
let ?x_rep = Rep_fixed_mat x
let ?y_rep = Rep_fixed_mat y
let ?c_rep = Rep_fixed_vec c
have c_mult_diff:  $c_{fv} * x - c_{fv} * y = Abs\_fixed\_vec (?c\_rep_v * (?x\_rep - ?y\_rep))$ 
proof -
have dim_col_eq_x_y: dim_col (Rep_fixed_mat x) = dim_col (Rep_fixed_mat y)
by (metis Rep_fixed_mat carrier_matD(2))
have vec_mult_x:  $c_{fv} * x = Abs\_fixed\_vec (?c\_rep_v * ?x\_rep)$ 
by (simp add: mult_vec_fixed_mat_def mult_vec_mat_def)
have vec_mult_y:  $c_{fv} * y = Abs\_fixed\_vec (?c\_rep_v * ?y\_rep)$ 
by (simp add: mult_vec_fixed_mat_def mult_vec_mat_def)
have  $c_{fv} * x - c_{fv} * y = Abs\_fixed\_vec (?c\_rep_v * ?x\_rep) - Abs\_fixed\_vec (?c\_rep_v * ?y\_rep)$ 
using vec_mult_x vec_mult_y by simp
also have ... = Abs_fixed_vec ((?c_rep_v * ?x_rep) - (?c_rep_v * ?y_rep))
proof -

```

```

have carrier_vx: (?c_rep v* ?x_rep) ∈ carrier_vec (CARD('nc))
  using Rep_fixed_vec
  by (metis mult_vec_fixed_mat.rep_eq mult_vec_mat_def)
have carrier_vy: (?c_rep v* ?y_rep) ∈ carrier_vec (CARD('nc))
  using Rep_fixed_vec
  by (metis mult_vec_fixed_mat.rep_eq mult_vec_mat_def)
show ?thesis
  using carrier_vx carrier_vy
  by (simp add: fixed_vec_add minus_add_uminus_vec pth_2 uminus_fixed_vec)
qed
also have ... = Abs_fixed_vec (?c_rep v* (?x_rep - ?y_rep))
proof -
  have (?c_rep v* ?x_rep) - (?c_rep v* ?y_rep) = ?c_rep v* (?x_rep - ?y_rep)
  using Rep_fixed_vec Rep_fixed_mat
  by (smt (verit, del_insts) Matrix.vec_eq_iff carrier_matD(1) col_dim col_minus_mat
    dim_col_eq_x_y dim_mult_vec_mat index_minus_mat(3) index_minus_vec(1)
    index_minus_vec(2) index_mult_vec_mat minus_scalar_prod_distrib)
  thus ?thesis by simp
qed
finally show ?thesis .
qed
have vec_index_eq:  $\forall i < \text{CARD}('nc).$ 
  fixed_vec_index (cfv* x - cfv* y) i = (?c_rep v* (?x_rep - ?y_rep)) $ i
proof (intro allI impI)
  fix i
  assume i < CARD('nc)
  show fixed_vec_index (cfv* x - cfv* y) i = (?c_rep v* (?x_rep - ?y_rep)) $ i
  using c_mult_diff
  by (metis (mono_tags, lifting) Abs_fixed_vec_inverse Rep_fixed_mat carrier_matD(2)
    fixed_vec_index.rep_eq index_minus_mat(3) mult_vec_mat_def vec_carrier)
qed
have vec_mult_expand:  $\forall i < \text{CARD}('nc).$ 
  (?c_rep v* (?x_rep - ?y_rep)) $ i = ( $\sum j = 0..<\text{CARD}('nr).$  ?c_rep $ j * (?x_rep - ?y_rep) $$ (j, i))
proof (intro allI impI)
  fix i
  assume i < CARD('nc)
  show (?c_rep v* (?x_rep - ?y_rep)) $ i = ( $\sum j = 0..<\text{CARD}('nr).$  ?c_rep $ j * (?x_rep - ?y_rep) $$ (j, i))
  proof -
    have carrier_c: ?c_rep ∈ carrier_vec (CARD('nr))
      by (rule Rep_fixed_vec)
    have carrier_diff: (?x_rep - ?y_rep) ∈ carrier_mat (CARD('nr)) (CARD('nc))
      using Rep_fixed_mat minus_carrier_mat by blast
    show ?thesis
      using carrier_c carrier_diff <i < CARD('nc)>
      by (metis (no_types, lifting) Finite_Cartesian_Product.sum_cong_aux carrier_matD(1)
        carrier_matD(2) index_vec_mat_mult)
  qed
qed
have triangle_step: ( $\sum i = 0..<\text{CARD}('nc).$  (norm (fixed_vec_index (cfv* x - cfv* y) i))2) ≤
  ( $\sum i = 0..<\text{CARD}('nc).$  ( $\sum j = 0..<\text{CARD}('nr).$  norm (?c_rep $ j) * norm ((?x_rep - ?y_rep) $$ (j, i))))2
proof -
  have  $\forall i < \text{CARD}('nc).$ 
  norm (fixed_vec_index (cfv* x - cfv* y) i) ≤
  ( $\sum j = 0..<\text{CARD}('nr).$  norm (?c_rep $ j * (?x_rep - ?y_rep) $$ (j, i)))

```

```

proof (intro alll impl)
  fix i
  assume i < CARD('nc)
  have norm (fixed_vec_index (cfv*x - cfv*y) i) =
    norm ((?c_repv* (?x_rep - ?y_rep)) $ i)
  using vec_index_eq ⟨i < CARD('nc)⟩ by simp
  also have ... = norm (∑ j = 0..<CARD('nr). ?c_rep $ j * (?x_rep - ?y_rep) $$ (j, i))
  using vec_mult_expand ⟨i < CARD('nc)⟩ by simp
  also have ... ≤ (∑ j = 0..<CARD('nr). norm (?c_rep $ j * (?x_rep - ?y_rep) $$ (j, i)))
  by (rule norm_sum)
  finally show norm (fixed_vec_index (cfv*x - cfv*y) i) ≤
    (∑ j = 0..<CARD('nr). norm (?c_rep $ j * (?x_rep - ?y_rep) $$ (j, i))).

qed
moreover have ∀ i < CARD('nc). ∀ j < CARD('nr).
  norm (?c_rep $ j * (?x_rep - ?y_rep) $$ (j, i)) =
  norm (?c_rep $ j) * norm ((?x_rep - ?y_rep) $$ (j, i))
  using norm_mult by blast
ultimately show ?thesis
proof -
  have ∀ i < CARD('nc).
    (norm (fixed_vec_index (cfv*x - cfv*y) i))2 ≤
    (∑ j = 0..<CARD('nr). norm (Rep_fixed_vec c $ j) * norm ((Rep_fixed_mat x - Rep_fixed_mat y) $$ (j, i)))2
  proof (intro alll impl)
    fix i
    assume i < CARD('nc)
    have norm (fixed_vec_index (cfv*x - cfv*y) i) ≤
      (∑ j = 0..<CARD('nr). norm (Rep_fixed_vec c $ j * (Rep_fixed_mat x - Rep_fixed_mat y) $$ (j, i)))
    using ⟨∀ i < CARD('nc). norm (fixed_vec_index (cfv*x - cfv*y) i) ≤ (∑ j = 0..<CARD('nr). norm (Rep_fixed_vec
  c $ j * (Rep_fixed_mat x - Rep_fixed_mat y) $$ (j, i)))⟩
    using ⟨i < CARD('nc)⟩ by blast
    also have ... = (∑ j = 0..<CARD('nr). norm (Rep_fixed_vec c $ j) * norm ((Rep_fixed_mat x - Rep_fixed_mat y)
  $$ (j, i)))
    using ⟨∀ i < CARD('nc). ∀ j < CARD('nr). norm (Rep_fixed_vec c $ j * (Rep_fixed_mat x - Rep_fixed_mat y) $$ (j, i))
  = norm (Rep_fixed_vec c $ j) * norm ((Rep_fixed_mat x - Rep_fixed_mat y) $$ (j, i))⟩
    using ⟨i < CARD('nc)⟩ by simp
    finally have norm (fixed_vec_index (cfv*x - cfv*y) i) ≤
      (∑ j = 0..<CARD('nr). norm (Rep_fixed_vec c $ j) * norm ((Rep_fixed_mat x - Rep_fixed_mat y) $$ (j, i)))
    .

    thus (norm (fixed_vec_index (cfv*x - cfv*y) i))2 ≤
      (∑ j = 0..<CARD('nr). norm (Rep_fixed_vec c $ j) * norm ((Rep_fixed_mat x - Rep_fixed_mat y) $$ (j, i)))2
    using norm_ge_zero power_mono by blast
  qed
thus ?thesis
  by (meson atLeastLessThan_iff sum_mono)
qed
qed
have cauchy_schwarz_step:
  (∑ i = 0..<CARD('nc). (∑ j = 0..<CARD('nr). norm (?c_rep $ j) * norm ((?x_rep - ?y_rep) $$ (j, i)))2) ≤
  (∑ i = 0..<CARD('nc).
    (∑ j = 0..<CARD('nr). (norm (?c_rep $ j))2) *
    (∑ j = 0..<CARD('nr). (norm ((?x_rep - ?y_rep) $$ (j, i)))2))
proof -
  have ∀ i < CARD('nc).
    (∑ j = 0..<CARD('nr). norm (?c_rep $ j) * norm ((?x_rep - ?y_rep) $$ (j, i)))2 ≤

```

```

( $\sum j = 0..<CARD('nr). (norm (?c\_rep \$ j))^2$ ) *
( $\sum j = 0..<CARD('nr). (norm ((?x\_rep - ?y\_rep) $$ (j, i))^2$ )
proof (intro alll impl)
  fix i
  assume  $i < CARD('nc)$ 
  let  $?a = \lambda j. norm (?c\_rep \$ j)$ 
  let  $?b = \lambda j. norm ((?x\_rep - ?y\_rep) $$ (j, i))$ 
  have  $nonneg\_a: \forall j. ?a j \geq 0$  by simp
  have  $nonneg\_b: \forall j. ?b j \geq 0$  by simp
  have  $(\sum j = 0..<CARD('nr). ?a j * ?b j)^2 \leq (\sum j = 0..<CARD('nr). (?a j)^2) * (\sum j = 0..<CARD('nr). (?b j)^2)$ 
    by (simp add: Cauchy_Schwarz_ineq_sum)
  thus  $(\sum j = 0..<CARD('nr). norm (?c\_rep \$ j) * norm ((?x\_rep - ?y\_rep) $$ (j, i))^2 \leq$ 
     $(\sum j = 0..<CARD('nr). (norm (?c\_rep \$ j))^2) * (\sum j = 0..<CARD('nr). (norm ((?x\_rep - ?y\_rep) $$ (j, i))^2)$ 
    by simp
  qed
thus ?thesis
  by (meson atLeastLessThan_iff sum_mono)
qed
have factor_step:
   $(\sum i = 0..<CARD('nc). (\sum j = 0..<CARD('nr). (norm (?c\_rep \$ j))^2) * (\sum j = 0..<CARD('nr). (norm ((?x\_rep - ?y\_rep) $$ (j, i))^2)) =$ 
   $(\sum j = 0..<CARD('nr). (norm (?c\_rep \$ j))^2) * (\sum i = 0..<CARD('nc). \sum j = 0..<CARD('nr). (norm ((?x\_rep - ?y\_rep) $$ (j, i))^2)$ 
  by (simp add: sum_distrib_left)
have norm_c_eq:  $(\sum j = 0..<CARD('nr). (norm (?c\_rep \$ j))^2) = (norm c)^2$ 
proof -
  have  $norm\ c = \sqrt{(\sum j = 0..<CARD('nr). (norm (fixed\_vec\_index\ c\ j))^2)}$ 
    unfolding norm_fixed_vec_def by blast
  also have  $\dots = \sqrt{(\sum j = 0..<CARD('nr). (norm (?c\_rep \$ j))^2)}$ 
    by (simp add: fixed_vec_index_def)
  finally have  $norm\ c = \sqrt{(\sum j = 0..<CARD('nr). (norm (?c\_rep \$ j))^2)}$ .
  thus ?thesis
    by (metis norm_ge_zero real_sqrt_ge_o_iff real_sqrt_pow2)
qed
have reorder_step:
   $(\sum i = 0..<CARD('nc). \sum j = 0..<CARD('nr). (norm ((?x\_rep - ?y\_rep) $$ (j, i))^2) =$ 
   $(\sum i = 0..<CARD('nr). \sum j = 0..<CARD('nc). (norm (fixed\_mat\_index\ (x - y)\ i\ j))^2)$ 
proof -
  have  $(\sum i = 0..<CARD('nc). \sum j = 0..<CARD('nr). (norm ((?x\_rep - ?y\_rep) $$ (j, i))^2) =$ 
     $(\sum j = 0..<CARD('nr). \sum i = 0..<CARD('nc). (norm ((?x\_rep - ?y\_rep) $$ (j, i))^2)$ 
    by (rule sum.swap)
  also have  $\dots = (\sum i = 0..<CARD('nr). \sum j = 0..<CARD('nc). (norm ((?x\_rep - ?y\_rep) $$ (i, j))^2)$ 
    by simp
  also have  $\dots = (\sum i = 0..<CARD('nr). \sum j = 0..<CARD('nc). (norm (fixed\_mat\_index\ (x - y)\ i\ j))^2)$ 
    proof -
    have  $\forall i < CARD('nr). \forall j < CARD('nc).$ 
       $norm ((Rep\_fixed\_mat\ x - Rep\_fixed\_mat\ y) $$ (i, j)) = norm (fixed\_mat\_index\ (x - y)\ i\ j)$ 
    proof (intro alll impl)
      fix i j
      assume  $i < CARD('nr)$  and  $j < CARD('nc)$ 
      have  $fixed\_mat\_index\ (x - y)\ i\ j = Rep\_fixed\_mat\ (x - y) \$\$ (i, j)$ 
        by (simp add: fixed_mat_index_def)
      also have  $\dots = (Rep\_fixed\_mat\ x - Rep\_fixed\_mat\ y) \$\$ (i, j)$ 

```

```

    using Rep_fixed_mat_minus by metis
    finally have fixed_mat_index (x - y) i j = (Rep_fixed_mat x - Rep_fixed_mat y) $$ (i, j) .
    thus norm ((Rep_fixed_mat x - Rep_fixed_mat y) $$ (i, j)) = norm (fixed_mat_index (x - y) i j)
    by simp
  qed
  thus ?thesis
  by simp
  qed
  then show ?thesis
  using <math>(\sum i = 0..<CARD('nc). \sum j = 0..<CARD('nr). (norm ((Rep_fixed_mat x - Rep_fixed_mat y) $$ (j, i)))^2) = (\sum j = 0..<CARD('nr). \sum i = 0..<CARD('nc). (norm ((Rep_fixed_mat x - Rep_fixed_mat y) $$ (j, i)))^2)>> by presburger
  qed
  have matrix_ineq: <math>(\sum i = 0..<CARD('nc). (norm (fixed_vec_index (c_{fv} * x - c_{fv} * y) i))^2) \leq (norm c)^2 * (\sum i = 0..<CARD('nr). \sum j = 0..<CARD('nc). (norm (fixed_mat_index (x - y) i j))^2)>
  using triangle_step cauchy_schwarz_step factor_step norm_c_eq reorder_step by simp
  have sqrt <math>(\sum i = 0..<CARD('nc). (norm (fixed_vec_index (c_{fv} * x - c_{fv} * y) i))^2) \leq \text{sqrt} ((norm c)^2 * (\sum i = 0..<CARD('nr). \sum j = 0..<CARD('nc). (norm (fixed_mat_index (x - y) i j))^2))>
  using matrix_ineq real_sqrt_le_iff by blast
  also have ... = norm c * sqrt <math>(\sum i = 0..<CARD('nr). \sum j = 0..<CARD('nc). (norm (fixed_mat_index (x - y) i j))^2)>
  proof -
    have sqrt <math>((norm c)^2 * (\sum i = 0..<CARD('nr). \sum j = 0..<CARD('nc). (norm (fixed_mat_index (x - y) i j))^2)) = \text{sqrt} ((norm c)^2) * \text{sqrt} (\sum i = 0..<CARD('nr). \sum j = 0..<CARD('nc). (norm (fixed_mat_index (x - y) i j))^2)>
    using real_sqrt_mult by blast
    also have sqrt <math>((norm c)^2) = norm c>
    by simp
    finally show ?thesis by simp
  qed
  also have ... \leq C * sqrt <math>(\sum i = 0..<CARD('nr). \sum j = 0..<CARD('nc). (norm (fixed_mat_index (x - y) i j))^2)>
  using C_bound C_nonneg
  by (metis mult_right_mono norm_fixed_mat_def norm_ge_zero)
  finally show sqrt <math>(\sum i = 0..<CARD('nc). (norm (fixed_vec_index (c_{fv} * x - c_{fv} * y) i))^2) \leq C * \text{sqrt} (\sum i = 0..<CARD('nr). \sum j = 0..<CARD('nc). (norm (fixed_mat_index (x - y) i j))^2)> .
  qed

```

lemma lipschitz_on_mat_mult:

assumes $\langle 0 \leq C \rangle$ **and** $\text{norm } c \leq C$

shows $\langle C\text{-lipschitz_on } U (\lambda (y::('a::\{\text{real_inner,real_normed_field}\}, 'nr::\text{finite}, 'nc::\text{finite}) \text{fixed_mat}) . (c::('a, 'nr) \text{fixed_vec})_{fv} * y) \rangle$

unfolding mult_vec_mat_def lipschitz_on_def

apply (intro conjl impl alll)

unfolding dist_fixed_vec_def dist_fixed_mat_def

unfolding norm_fixed_vec_def norm_fixed_mat_def

subgoal using assms **by** blast

subgoal using vector_matrix_inequality[of c C U] assms **by** blast

done

lemma lipschitz_on_weighted_sum_bias:

assumes $\langle 0 \leq C \rangle$ **and** $\text{norm } c \leq C$

shows $\langle C\text{-lipschitz_on } U (\lambda (y::('a::\{\text{real_inner,real_normed_field}\}, 'nr::\text{finite}, 'nc::\text{finite}) \text{fixed_mat}) . (c_{fv} * y) + b) \rangle$

using lipschitz_on_mat_mult

proof -

```

have  $\forall F. C\text{--lipschitz\_on } (F::('a, 'nr, 'nc) \text{fixed\_mat set}) ((f_{v*}) c)$ 
  by (meson assms(2) lipschitz_on_le lipschitz_on_mat_mult norm_imp_pos_and_ge)
then have  $\forall F f. (o + C)\text{--lipschitz\_on } F (\lambda fa. (f::('a, 'nc) \text{fixed\_vec}) + c_{f_{v*}} fa)$ 
  using Lipschitz.lipschitz_on_constant lipschitz_on_add by blast
then have  $C\text{--lipschitz\_on } U (\lambda f. b + c_{f_{v*}} f)$ 
  by simp
then show ?thesis
  by (meson Groups.add_ac(2) lipschitz_on_cong)
qed

```

lemma mult_vec_mat_distrib_left:

```

assumes  $v1 \in \text{carrier\_vec } n$  and  $v2 \in \text{carrier\_vec } n$  and  $A \in \text{carrier\_mat } n \ m$ 
shows  $(v1 - v2)_{v*} A = v1_{v*} A - v2_{v*} A$  ( $A:: 'a::\{\text{real\_normed\_field, real\_inner}\} \text{mat}$ )

```

proof -

```

have carrier_diff:  $v1 - v2 \in \text{carrier\_vec } n$ 
  using assms by simp

```

```

have carrier_mult1:  $v1_{v*} A \in \text{carrier\_vec } m$ 
  using assms

```

```

by (simp add: mult_vec_mat_def)

```

```

have carrier_mult2:  $v2_{v*} A \in \text{carrier\_vec } m$ 
  using assms

```

```

by (simp add: mult_vec_mat_def)

```

```

have carrier_mult_diff:  $(v1 - v2)_{v*} A \in \text{carrier\_vec } m$ 
  using carrier_diff assms

```

```

using carrier_dim_vec dim_mult_vec_mat by blast

```

```

have dim_vec  $((v1 - v2)_{v*} A) = \text{dim\_vec } (v1_{v*} A - v2_{v*} A)$ 
  using carrier_mult_diff carrier_mult1 carrier_mult2 by simp

```

```

show ?thesis

```

proof

```

show dim_vec  $((v1 - v2)_{v*} A) = \text{dim\_vec } (v1_{v*} A - v2_{v*} A)$ 
  by (simp add: mult_vec_mat_def)

```

next

fix i

```

assume i_bound:  $i < \text{dim\_vec } (v1_{v*} A - v2_{v*} A)$ 

```

```

have i_less_m:  $i < m$ 

```

```

using i_bound carrier_mult1 carrier_mult2 by simp

```

```

have lhs_expand:  $((v1 - v2)_{v*} A) \$ i = (\sum j = 0..<n. (v1 - v2) \$ j * A \$\$ (j, i))$ 

```

```

using carrier_diff assms i_less_m mult_vec_mat_def

```

```

by (metis (no_types, lifting) carrier_matD(1) carrier_matD(2) index_vec_mat_mult sum.cong)

```

```

have vec_sub_expand:  $(\sum j = 0..<n. (v1 - v2) \$ j * A \$\$ (j, i)) = (\sum j = 0..<n. (v1 \$ j - v2 \$ j) * A \$\$ (j, i))$ 

```

```

using assms by simp

```

```

have mult_distrib:  $(\sum j = 0..<n. (v1 \$ j - v2 \$ j) * A \$\$ (j, i)) = (\sum j = 0..<n. v1 \$ j * A \$\$ (j, i) - v2 \$ j * A \$\$ (j, i))$ 

```

```

by (meson vector_space_over_itself.scale_left_diff_distrib)

```

```

have sum_distrib:  $(\sum j = 0..<n. v1 \$ j * A \$\$ (j, i) - v2 \$ j * A \$\$ (j, i)) = (\sum j = 0..<n. v1 \$ j * A \$\$ (j, i)) - (\sum j = 0..<n. v2 \$ j * A \$\$ (j, i))$ 

```

```

by (simp add: sum_subtractf)

```

```

have back_to_mult:  $(\sum j = 0..<n. v1 \$ j * A \$\$ (j, i)) - (\sum j = 0..<n. v2 \$ j * A \$\$ (j, i)) = (v1_{v*} A) \$ i - (v2_{v*} A) \$ i$ 

```

```

by (metis (mono_tags, lifting) assms(1) assms(2) assms(3) carrier_matD(1) carrier_matD(2) i_less_m index_vec_mat_mult)

```

```

have to_vec_sub:  $(v1_{v*} A) \$ i - (v2_{v*} A) \$ i = (v1_{v*} A - v2_{v*} A) \$ i$ 

```

```

using carrier_mult1 carrier_mult2 i_less_m by simp

```

```

show  $((v1 - v2)_{v*} A) \$ i = (v1_{v*} A - v2_{v*} A) \$ i$ 

```

using lhs_expand vec_sub_expand mult_distrib sum_distrib back_to_mult to_vec_sub by simp
qed
qed

lemma matrix_vector_inequality:

fixes c :: ('a::{real_normed_field,real_inner}, 'nr::finite, 'nc::finite) fixed_mat
and U :: ('a, 'nr) fixed_vec set
and C :: real

assumes C_bound: $C \geq \text{norm } c$
and C_nonneg: $C \geq 0$

shows $\bigwedge x y. x \in U \implies y \in U \implies$

$\text{sqrt} \left(\sum_{i=0..<\text{CARD}('nc)}. (\text{norm} (\text{fixed_vec_index } (x_{fv} * c - y_{fv} * c) i))^2 \right) \leq$
 $C * \text{sqrt} \left(\sum_{i=0..<\text{CARD}('nr)}. (\text{norm} (\text{fixed_vec_index } (x - y) i))^2 \right)$

proof -

fix x y :: ('a, 'nr) fixed_vec

assume x_in_U: $x \in U$ and y_in_U: $y \in U$

let ?x_rep = Rep_fixed_vec x

let ?y_rep = Rep_fixed_vec y

let ?c_rep = Rep_fixed_mat c

have vec_mat_mult_diff: $x_{fv} * c - y_{fv} * c = \text{Abs_fixed_vec } (?x_{rep} v * ?c_{rep} - ?y_{rep} v * ?c_{rep})$

proof -

have vec_mat_mult_x: $x_{fv} * c = \text{Abs_fixed_vec } (?x_{rep} v * ?c_{rep})$

by (simp add: mult_vec_fixed_mat_def mult_vec_mat_def)

have vec_mat_mult_y: $y_{fv} * c = \text{Abs_fixed_vec } (?y_{rep} v * ?c_{rep})$

by (simp add: mult_vec_fixed_mat_def mult_vec_mat_def)

have $x_{fv} * c - y_{fv} * c = \text{Abs_fixed_vec } (?x_{rep} v * ?c_{rep}) - \text{Abs_fixed_vec } (?y_{rep} v * ?c_{rep})$

using vec_mat_mult_x vec_mat_mult_y by simp

also have ... = $\text{Abs_fixed_vec } ((?x_{rep} v * ?c_{rep}) - (?y_{rep} v * ?c_{rep}))$

proof -

have carrier_vx: $(?x_{rep} v * ?c_{rep}) \in \text{carrier_vec } (\text{CARD}('nc))$

using Rep_fixed_vec

by (metis mult_vec_fixed_mat.rep_eq mult_vec_mat_def)

have carrier_vy: $(?y_{rep} v * ?c_{rep}) \in \text{carrier_vec } (\text{CARD}('nc))$

using Rep_fixed_vec

by (metis mult_vec_fixed_mat.rep_eq mult_vec_mat_def)

show ?thesis

using carrier_vx carrier_vy

by (simp add: fixed_vec_add minus_add_uminus_vec pth_2 uminus_fixed_vec)

qed

finally show ?thesis by simp

qed

have vec_mat_mult_linear: $?x_{rep} v * ?c_{rep} - ?y_{rep} v * ?c_{rep} = (?x_{rep} - ?y_{rep}) v * ?c_{rep}$

proof -

have $?x_{rep} v * ?c_{rep} - ?y_{rep} v * ?c_{rep} = (?x_{rep} - ?y_{rep}) v * ?c_{rep}$

using Rep_fixed_vec Rep_fixed_mat

using mult_vec_mat_distrib_left

by metis

thus ?thesis by simp

qed

have c_mult_diff: $x_{fv} * c - y_{fv} * c = \text{Abs_fixed_vec } ((?x_{rep} - ?y_{rep}) v * ?c_{rep})$

using vec_mat_mult_diff vec_mat_mult_linear by simp

have vec_index_eq: $\forall i < \text{CARD}('nc).$

$\text{fixed_vec_index } (x_{fv} * c - y_{fv} * c) i = ((?x_{rep} - ?y_{rep}) v * ?c_{rep}) \$ i$

proof (intro allI impI)

```

fix i
assume i < CARD('nc)
show fixed_vec_index (x_{fv} * c - y_{fv} * c) i = ((?x_rep - ?y_rep)_{v} * ?c_rep) $ i
  using c_mult_diff
  by (metis (mono_tags, lifting) Abs_fixed_vec_inverse Rep_fixed_mat carrier_matD(2)
      fixed_vec_index.rep_eq mult_vec_mat_def vec_carrier)
qed
have vec_mat_expand:  $\forall i < \text{CARD}('nc).$ 
   $((?x_{\text{rep}} - ?y_{\text{rep}})_{v} * ?c_{\text{rep}}) \$ i = (\sum j = 0..<\text{CARD}('nr). (?x_{\text{rep}} - ?y_{\text{rep}}) \$ j * ?c_{\text{rep}} \$\$ (j, i))$ 
proof (intro allI impI)
fix i
assume i < CARD('nc)
show  $((?x_{\text{rep}} - ?y_{\text{rep}})_{v} * ?c_{\text{rep}}) \$ i = (\sum j = 0..<\text{CARD}('nr). (?x_{\text{rep}} - ?y_{\text{rep}}) \$ j * ?c_{\text{rep}} \$\$ (j, i))$ 
proof -
  have carrier_diff:  $(?x_{\text{rep}} - ?y_{\text{rep}}) \in \text{carrier\_vec} (\text{CARD}('nr))$ 
  using Rep_fixed_vec
  using minus_carrier_vec by blast
  have carrier_c:  $?c_{\text{rep}} \in \text{carrier\_mat} (\text{CARD}('nr)) (\text{CARD}('nc))$ 
  by (rule Rep_fixed_mat)
  show ?thesis
  using carrier_diff carrier_c <i < CARD('nc)>
  mult_vec_mat_def scalar_prod_def
  by (simp add: index_vec_mat_mult)
qed
qed
have triangle_step:  $(\sum i = 0..<\text{CARD}('nc). (\text{norm} (\text{fixed\_vec\_index} (x_{fv} * c - y_{fv} * c) i))^2) \leq$ 
 $(\sum i = 0..<\text{CARD}('nc). (\sum j = 0..<\text{CARD}('nr). \text{norm} ((?x_{\text{rep}} - ?y_{\text{rep}}) \$ j) * \text{norm} (?c_{\text{rep}} \$\$ (j, i))))^2$ 
proof -
  have  $\forall i < \text{CARD}('nc).$ 
   $\text{norm} (\text{fixed\_vec\_index} (x_{fv} * c - y_{fv} * c) i) \leq$ 
   $(\sum j = 0..<\text{CARD}('nr). \text{norm} ((?x_{\text{rep}} - ?y_{\text{rep}}) \$ j * ?c_{\text{rep}} \$\$ (j, i)))$ 
proof (intro allI impI)
fix i
assume i < CARD('nc)
have  $\text{norm} (\text{fixed\_vec\_index} (x_{fv} * c - y_{fv} * c) i) =$ 
   $\text{norm} (((?x_{\text{rep}} - ?y_{\text{rep}})_{v} * ?c_{\text{rep}}) \$ i)$ 
  using vec_index_eq <i < CARD('nc)> by simp
also have  $\dots = \text{norm} (\sum j = 0..<\text{CARD}('nr). (?x_{\text{rep}} - ?y_{\text{rep}}) \$ j * ?c_{\text{rep}} \$\$ (j, i))$ 
  using vec_mat_expand <i < CARD('nc)> by simp
also have  $\dots \leq (\sum j = 0..<\text{CARD}('nr). \text{norm} ((?x_{\text{rep}} - ?y_{\text{rep}}) \$ j * ?c_{\text{rep}} \$\$ (j, i)))$ 
  by (rule norm_sum)
finally show  $\text{norm} (\text{fixed\_vec\_index} (x_{fv} * c - y_{fv} * c) i) \leq$ 
 $(\sum j = 0..<\text{CARD}('nr). \text{norm} ((?x_{\text{rep}} - ?y_{\text{rep}}) \$ j * ?c_{\text{rep}} \$\$ (j, i))).$ 
qed
moreover have  $\forall i < \text{CARD}('nc). \forall j < \text{CARD}('nr).$ 
 $\text{norm} ((?x_{\text{rep}} - ?y_{\text{rep}}) \$ j * ?c_{\text{rep}} \$\$ (j, i)) =$ 
 $\text{norm} ((?x_{\text{rep}} - ?y_{\text{rep}}) \$ j) * \text{norm} (?c_{\text{rep}} \$\$ (j, i))$ 
  using norm_mult by blast
ultimately show ?thesis
proof -
  have  $\forall i < \text{CARD}('nc).$ 
   $(\text{norm} (\text{fixed\_vec\_index} (x_{fv} * c - y_{fv} * c) i))^2 \leq$ 
   $(\sum j = 0..<\text{CARD}('nr). \text{norm} ((\text{Rep\_fixed\_vec } x - \text{Rep\_fixed\_vec } y) \$ j) * \text{norm} (\text{Rep\_fixed\_mat } c \$\$ (j, i))))^2$ 
proof (intro allI impI)

```

```

fix i
assume i < CARD('nc)
have norm (fixed_vec_index (x_{fv}*c - y_{fv}*c) i) ≤
  (∑ j = 0..<CARD('nr). norm ((Rep_fixed_vec x - Rep_fixed_vec y) $ j * Rep_fixed_mat c $$ (j, i)))
using <∀ i < CARD('nc). norm (fixed_vec_index (x_{fv}*c - y_{fv}*c) i) ≤ (∑ j = 0..<CARD('nr). norm ((Rep_fixed_vec
x - Rep_fixed_vec y) $ j * Rep_fixed_mat c $$ (j, i)))>
  using <i < CARD('nc)> by blast
also have ... = (∑ j = 0..<CARD('nr). norm ((Rep_fixed_vec x - Rep_fixed_vec y) $ j) * norm (Rep_fixed_mat c $$
(j, i)))
  using <∀ i < CARD('nc). ∀ j < CARD('nr). norm ((Rep_fixed_vec x - Rep_fixed_vec y) $ j * Rep_fixed_mat c $$ (j, i))
= norm ((Rep_fixed_vec x - Rep_fixed_vec y) $ j) * norm (Rep_fixed_mat c $$ (j, i))>
  using <i < CARD('nc)> by simp
finally have norm (fixed_vec_index (x_{fv}*c - y_{fv}*c) i) ≤
  (∑ j = 0..<CARD('nr). norm ((Rep_fixed_vec x - Rep_fixed_vec y) $ j) * norm (Rep_fixed_mat c $$ (j, i))).
thus (norm (fixed_vec_index (x_{fv}*c - y_{fv}*c) i))^2 ≤
  (∑ j = 0..<CARD('nr). norm ((Rep_fixed_vec x - Rep_fixed_vec y) $ j) * norm (Rep_fixed_mat c $$ (j, i)))^2
  using norm_ge_zero power_mono by blast
qed
thus ?thesis
  by (meson atLeastLessThan_iff sum_mono)
qed
qed
have cauchy_schwarz_step:
  (∑ i = 0..<CARD('nc). (∑ j = 0..<CARD('nr). norm ((?x_rep - ?y_rep) $ j) * norm (?c_rep $$ (j, i)))^2) ≤
  (∑ i = 0..<CARD('nc).
    (∑ j = 0..<CARD('nr). (norm ((?x_rep - ?y_rep) $ j))^2) *
    (∑ j = 0..<CARD('nr). (norm (?c_rep $$ (j, i)))^2))
proof -
have ∀ i < CARD('nc).
  (∑ j = 0..<CARD('nr). norm ((?x_rep - ?y_rep) $ j) * norm (?c_rep $$ (j, i)))^2 ≤
  (∑ j = 0..<CARD('nr). (norm ((?x_rep - ?y_rep) $ j))^2) *
  (∑ j = 0..<CARD('nr). (norm (?c_rep $$ (j, i)))^2)
proof (intro allI impl)
fix i
assume i < CARD('nc)
let ?a = λj. norm ((?x_rep - ?y_rep) $ j)
let ?b = λj. norm (?c_rep $$ (j, i))
have nonneg_a: ∀ j. ?a j ≥ 0 by simp
have nonneg_b: ∀ j. ?b j ≥ 0 by simp
have (∑ j = 0..<CARD('nr). ?a j * ?b j)^2 ≤ (∑ j = 0..<CARD('nr). (?a j)^2) * (∑ j = 0..<CARD('nr). (?b j)^2)
  by (simp add: Cauchy_Schwarz_ineq_sum)
thus (∑ j = 0..<CARD('nr). norm ((?x_rep - ?y_rep) $ j) * norm (?c_rep $$ (j, i)))^2 ≤
  (∑ j = 0..<CARD('nr). (norm ((?x_rep - ?y_rep) $ j))^2) * (∑ j = 0..<CARD('nr). (norm (?c_rep $$ (j, i)))^2)
  by simp
qed
thus ?thesis by (meson atLeastLessThan_iff sum_mono)
qed
qed
have factor_step:
  (∑ i = 0..<CARD('nc).
    (∑ j = 0..<CARD('nr). (norm ((?x_rep - ?y_rep) $ j))^2) *
    (∑ j = 0..<CARD('nr). (norm (?c_rep $$ (j, i)))^2)) =
  (∑ j = 0..<CARD('nr). (norm ((?x_rep - ?y_rep) $ j))^2) *
  (∑ i = 0..<CARD('nc). ∑ j = 0..<CARD('nr). (norm (?c_rep $$ (j, i)))^2)
  by (simp add: sum_distrib_left)

```

```

have norm_vec_diff_eq: ( $\sum j = 0..<CARD('nr). (norm ((?x_rep - ?y_rep) \$ j))^2$ ) = ( $norm (x - y)$ )2
proof -
  have  $norm (x - y) = sqrt (\sum j = 0..<CARD('nr). (norm (fixed_vec_index (x - y) j))^2)$ 
    unfolding norm_fixed_vec_def by blast
  also have ... =  $sqrt (\sum j = 0..<CARD('nr). (norm ((?x_rep - ?y_rep) \$ j))^2)$ 
  proof -
    have  $\forall j < CARD('nr). fixed\_vec\_index (x - y) j = (?x\_rep - ?y\_rep) \$ j$ 
    proof (intro allI impI)
      fix j
      assume  $j < CARD('nr)$ 
      show  $fixed\_vec\_index (x - y) j = (?x\_rep - ?y\_rep) \$ j$ 
      proof -
        have  $fixed\_vec\_index (x - y) j = Rep\_fixed\_vec (x - y) \$ j$ 
        by (simp add: fixed_vec_index_def)
        also have ... =  $(?x\_rep - ?y\_rep) \$ j$ 
        by (smt (verit, ccfv_threshold) Abs_fixed_vec_inverse Matrix.minus_vec_def
            Rep_fixed_vec carrier_vecD fixed_vec_add minus_add uminus_vec minus_fixed_vec.rep_eq
            pth_2 uminus_fixed_vec vec_carrier zero_minus_vec)
        finally show ?thesis .
      qed
    qed
  thus ?thesis by simp
  qed
  finally have  $norm (x - y) = sqrt (\sum j = 0..<CARD('nr). (norm ((?x_rep - ?y_rep) \$ j))^2)$  .
  thus ?thesis by (metis norm_ge_zero real_sqrt_ge_o_iff real_sqrt_pow2)
  qed
have norm_mat_eq: ( $\sum i = 0..<CARD('nc). \sum j = 0..<CARD('nr). (norm (?c_rep \$\$ (j, i)))^2$ ) = ( $norm c$ )2
proof -
  have ( $\sum i = 0..<CARD('nc). \sum j = 0..<CARD('nr). (norm (?c_rep \$\$ (j, i)))^2$ ) =
    ( $\sum j = 0..<CARD('nr). \sum i = 0..<CARD('nc). (norm (?c_rep \$\$ (j, i)))^2$ )
    by (rule sum.swap)
  also have ... = ( $\sum i = 0..<CARD('nr). \sum j = 0..<CARD('nc). (norm (fixed_mat_index c i j))^2$ )
  proof -
    have  $\forall i < CARD('nr). \forall j < CARD('nc).$ 
       $norm (?c\_rep \$\$ (i, j)) = norm (fixed\_mat\_index c i j)$ 
    proof (intro allI impI)
      fix i j
      assume  $i < CARD('nr)$  and  $j < CARD('nc)$ 
      have  $fixed\_mat\_index c i j = ?c\_rep \$\$ (i, j)$ 
      by (simp add: fixed_mat_index_def)
      thus  $norm (?c\_rep \$\$ (i, j)) = norm (fixed\_mat\_index c i j)$ 
      by simp
    qed
  thus ?thesis by simp
  qed
  also have ... = ( $norm c$ )2
  proof -
    have  $norm c = sqrt (\sum i = 0..<CARD('nr). \sum j = 0..<CARD('nc). (norm (fixed_mat_index c i j))^2)$ 
    unfolding norm_fixed_mat_def by blast
    thus ?thesis by (metis norm_ge_zero real_sqrt_ge_o_iff real_sqrt_pow2)
  qed
  finally show ?thesis .
  qed
have matrix_ineq: ( $\sum i = 0..<CARD('nc). (norm (fixed_vec_index (x_{fv} * c - y_{fv} * c) i))^2$ )  $\leq$ 

```

```

(norm (x - y))2 * (norm c)2
using triangle_step cauchy_schwarz_step factor_step norm_vec_diff_eq norm_mat_eq by simp
have sqrt (∑ i = 0.. $\text{CARD}('nc)$ . (norm (fixed_vec_index (xfv*c - yfv*c) i))2) ≤
  sqrt ((norm (x - y))2 * (norm c)2)
using matrix_ineq
using real_sqrt_le_iff by blast
also have ... = norm (x - y) * norm c
proof -
have sqrt ((norm (x - y))2 * (norm c)2) = sqrt ((norm (x - y))2) * sqrt ((norm c)2)
  by (rule real_sqrt_mult)
also have ... = norm (x - y) * norm c
  by simp
finally show ?thesis by simp
qed
also have ... = norm c * norm (x - y)
  by (simp add: mult.commute)
also have ... ≤ C * norm (x - y)
  using C_bound C_nonneg by (simp add: mult_right_mono norm_ge_zero)
also have ... = C * sqrt ((norm (x - y))2)
  by simp
also have ... = C * sqrt (∑ i = 0.. $\text{CARD}('nr)$ . (norm (fixed_vec_index (x - y) i))2)
proof -
have norm (x - y) = sqrt (∑ i = 0.. $\text{CARD}('nr)$ . (norm (fixed_vec_index (x - y) i))2)
  unfolding norm_fixed_vec_def by blast
thus ?thesis
  using ⟨(C::real) * norm ((x::('a, 'nr) fixed_vec) - (y::('a, 'nr) fixed_vec)) = C * sqrt ((norm (x - y))2)⟩ by presburger
qed
finally show sqrt (∑ i = 0.. $\text{CARD}('nc)$ . (norm (fixed_vec_index (xfv*c - yfv*c) i))2) ≤
  C * sqrt (∑ i = 0.. $\text{CARD}('nr)$ . (norm (fixed_vec_index (x - y) i))2).
qed

```

lemma lipschitz_on_vec_mult:

```

assumes ⟨0 ≤ C⟩ and norm c ≤ C
shows ⟨C-lipschitz_on U (λ y . yfv*c * (c::('a:: {real_inner,real_normed_field}, 'nr::finite, 'nc::finite) fixed_mat))⟩
unfolding mult_vec_mat_def lipschitz_on_def
apply (intro conj1 impl all1)
unfolding dist_fixed_vec_def dist_fixed_mat_def
unfolding norm_fixed_vec_def norm_fixed_mat_def
subgoal using assms by blast
subgoal using matrix_vector_inequality[of c C U _]
  using assms(1) assms(2) by blast
done

```

lemma C_ge_norm:

```

norm (c::('a:: {real_algebra_1,real_normed_vector}, 'nr::finite, 'nc::finite) fixed_mat) ≤ C ⇒ 0 ≤ C
by (smt (verit, ccfv_SIG) norm_ge_zero)

```

lemma lipschitz_on_weighted_sum_bias':

```

assumes norm c ≤ C
shows ⟨C-lipschitz_on U (λ y . (yfv*c * (c::('a:: {real_inner,real_normed_field}, 'nr::finite, 'nc::finite) fixed_mat)) +
b)⟩
using lipschitz_on_vec_mult assms C_ge_norm[of c C, simplified assms, simplified]
using Lipschitz.lipschitz_on_constant lipschitz_on_add
by fastforce

```

lemma lipschitz_on_dense:
assumes $\text{norm } c \leq C$
assumes $D\text{-lipschitz_on } ((\lambda y. (c::('a::\{\text{real_inner,real_normed_field}\}, 'n::\text{finite}) \text{fixed_vec})_{fv} * y + b) 'U) f$
shows $(D * C)\text{-lipschitz_on } U (\lambda y. f ((c_{fv} * y) + b))$
using $\text{lipschitz_on_weighted_sum_bias[of } D \ c]$
using $\text{lipschitz_on_compose2[of } D \ U \ \lambda y. (c_{fv} * y) + b \ C \ f, \text{ simplified assms}]$
using $\text{assms(1) assms(2) lipschitz_on_compose2 lipschitz_on_weighted_sum_bias}$
proof –
have $0 \leq C$
by $(\text{meson assms(1) landau_o.R_trans norm_ge_zero})$
then show $?thesis$
using $\text{assms(1) assms(2) lipschitz_on_compose2 lipschitz_on_weighted_sum_bias by blast}$
qed

lemma lipschitz_on_dense':
assumes $\text{norm } c \leq C$
assumes $D\text{-lipschitz_on } ((\lambda y. y_{fv} * (c::('a::\{\text{real_inner,real_normed_field}\}, 'nr::\text{finite}, 'nc::\text{finite}) \text{fixed_mat}) + b) 'U) f$
shows $(D * C)\text{-lipschitz_on } U (\lambda y. f ((y_{fv} * c) + b))$
using $\text{lipschitz_on_weighted_sum_bias[of } D \ _ \ b]$
using $\text{lipschitz_on_compose2[of } D \ U \ \lambda y. (y_{fv} * c) + b \ C \ f, \text{ simplified assms}]$
using $\text{assms(1) assms(2) lipschitz_on_compose2 lipschitz_on_weighted_sum_bias}$
using $\text{lipschitz_on_weighted_sum_bias' by blast}$

lemma lipschitz_on_input_output:
shows $1\text{-lipschitz_on } U (\lambda i. i)$
unfolding $\text{lipschitz_on_def by simp}$

lemma lipschitz_on_activation:
assumes $C\text{-lipschitz_on } U f$
shows $C\text{-lipschitz_on } U (\lambda i. f i)$
using assms by simp

Neural Network: Layers **fun** $\text{predict}_{\text{layer}_f} :: ((('a::\{\text{real_algebra}_1, \text{real_normed_vector}\}, 'b::\text{finite}) \text{fixed_vec}, 'c, 'd) \text{neural_network_seq_layers} \Rightarrow ('a, 'b::\text{finite}) \text{fixed_vec} \Rightarrow (('a, 'b) \text{fixed_vec}, 'c, ('a, 'b, 'b) \text{fixed_mat}) \text{layer} \Rightarrow ('a, 'b) \text{fixed_vec})$
where

$\langle \text{predict}_{\text{layer}_f} N (vs) (\text{In } l) = vs \rangle$
 $\langle \text{predict}_{\text{layer}_f} N (vs) (\text{Out } l) = vs \rangle$
 $\langle \text{predict}_{\text{layer}_f} N (vs) (\text{Dense } pl) = ((\text{the } (\text{activation_tab } N (\varphi pl))) ((vs_{fv} * \omega pl) + \beta pl)) \rangle$
 $\langle \text{predict}_{\text{layer}_f} N (vs) (\text{Activation } pl) = ((\text{the } (\text{activation_tab } N (\varphi pl))) vs) \rangle$

definition $\langle \text{predict}_{\text{seq_layer}_f} N \text{ inputs} = \text{foldl } (\text{predict}_{\text{layer}_f} N) \text{ inputs } (\text{layers } N) \rangle$

definition $\langle \text{lipschitz_continuous_activation_tab}_f \text{ tab } U = (\forall f \in \text{ran } (\text{tab}). \exists C. C\text{-lipschitz_on } U f) \rangle$

lemma input_layer_lipschitz_on:
 $1\text{-lipschitz_on } U ((\lambda vs. (\text{predict}_{\text{layer}_f} N vs (\text{In } x1))))$
by $(\text{simp add: lipschitz_on_def})$

lemma *output_layer_lipschitz_on*:

$1\text{-lipschitz_on } U ((\lambda vs . (predict_{layer_f} N vs (Out x1))))$
by (*simp add: lipschitz_on_def*)

lemma *dense_layer_lipschitz_on*:

assumes $norm (\omega x1) \leq C$
assumes $D\text{-lipschitz_on } ((\lambda y. y_{fv} * \omega x1 + \beta x1) ' U) (the (activation_tab N (\varphi x1)))$
shows $(C * D)\text{-lipschitz_on } U (\lambda vs :: ('a :: \{real_inner, real_normed_field\}, 'c :: finite) fixed_vec . (predict_{layer_f} N vs (Dense x1)))$
using *predict_{layer_f}.simps(3)[of N _ x1]*
using *lipschitz_on_dense'[of $\omega x1 C D \beta x1 U$, simplified assms]*
by (*metis (no_types, lifting) assms(2) lipschitz_on_transform mult_commute_abs*)

lemma *activation_layer_lipschitz_on*:

assumes $C\text{-lipschitz_on } U (the (activation_tab N (\varphi x1)))$
shows $C\text{-lipschitz_on } U (\lambda vs . (predict_{layer_f} N vs (Activation x1)))$
using *assms by simp*

lemma *foldl_layer_lipschitz_on*:

fixes $N :: (('a :: \{real_algebra_1, real_normed_vector\}, 'b :: finite) fixed_vec, 'c, 'd) neural_network_seq_layers$
assumes *layer_lipschitz*: $\forall l \in set\ ls. \exists C. C\text{-lipschitz_on } U (\lambda vs. predict_{layer_f} N vs l)$
assumes *domain_invariant*: $\forall l \in set\ ls. \forall vs \in U. predict_{layer_f} N vs l \in U$
shows $\exists C. C\text{-lipschitz_on } U (\lambda vs. foldl (predict_{layer_f} N) vs ls)$
using *assms*

proof (*induction ls*)

case *Nil*

then show ?*case*

by (*auto simp: lipschitz_on_def intro: ex1[where x=1]*)

next

case (*Cons l ls*)

have $l \in set (l \# ls)$ **using** *Cons by simp*

obtain $C1$ **where** $h1: C1\text{-lipschitz_on } U (\lambda vs. predict_{layer_f} N vs l)$

using *Cons(2) by auto*

obtain $C2$ **where** $h2: C2\text{-lipschitz_on } U (\lambda vs. foldl (predict_{layer_f} N) vs ls)$

using *Cons by auto*

have *fold_step*: $\forall vs. foldl (predict_{layer_f} N) vs (l \# ls) =$
 $foldl (predict_{layer_f} N) (predict_{layer_f} N vs l) ls$

by *simp*

have *composition*: $\forall vs. foldl (predict_{layer_f} N) vs (l \# ls) =$
 $(\lambda x. foldl (predict_{layer_f} N) x ls) (predict_{layer_f} N vs l)$

using *fold_step by simp*

have *image_in_domain*: $(\lambda vs. predict_{layer_f} N vs l) ' U \subseteq U$

using *Cons(1) h1 h2*

using *l local.Cons(3) by fastforce*

have $(C1 * C2)\text{-lipschitz_on } U (\lambda vs. foldl (predict_{layer_f} N) vs (l \# ls))$

using *lipschitz_on_compose[of C1 U ($\lambda vs. predict_{layer_f} N vs l$) C2 ($\lambda vs. foldl (predict_{layer_f} N) vs ls$), simplified h1, simplified]*

image_in_domain composition

proof —

have $(C2 * C1)\text{-lipschitz_on } U (\lambda f. foldl (predict_{layer_f} N) f (l \# ls))$

using $\langle C2\text{-lipschitz_on } ((\lambda vs. predict_{layer_f} N vs l) ' U) (\lambda vs. foldl (predict_{layer_f} N) vs ls) \implies (C2 * C1)\text{-lipschitz_on } U (\lambda x. foldl (predict_{layer_f} N) (predict_{layer_f} N x l) ls) \rangle h2$ *image_in_domain lipschitz_on_subset*

by *force*

then show ?*thesis*

by argo
qed
thus ?case by auto
qed

lemma layers_lipschitz_from_components:

fixes N :: (('a::{real_algebra_1,real_normed_vector,real_inner,real_normed_field}, 'b::finite) fixed_vec, 'c, 'd) neural_network_seq_layers

assumes weight_bounded: $\forall l \in \text{set } ls. (\text{case } l \text{ of Dense } pl \Rightarrow \text{norm } (\omega \text{ } pl) \leq W)$

assumes activation_lipschitz: $\forall l \in \text{set } ls. (\text{case } l \text{ of}$

Dense $pl \Rightarrow \exists D. D\text{-lipschitz_on } ((\lambda y. y_{fv} * \omega \text{ } pl + \beta \text{ } pl) ' U) (\text{the } (\text{activation_tab } N (\varphi \text{ } pl))) \mid$

Activation $pl \Rightarrow \exists C. C\text{-lipschitz_on } U (\text{the } (\text{activation_tab } N (\varphi \text{ } pl)))$)

shows $\forall l \in \text{set } ls. \exists C. C\text{-lipschitz_on } U (\lambda vs. \text{predict}_{layer_f} N \text{ vs } l)$

proof

fix l assume l ∈ set ls

show $\exists C. C\text{-lipschitz_on } U (\lambda vs. \text{predict}_{layer_f} N \text{ vs } l)$

proof (cases l)

case (In x)

then show ?thesis using input_layer_lipschitz_on by blast

next

case (Out x)

then show ?thesis using output_layer_lipschitz_on by blast

next

case (Dense x)

obtain D where $D\text{-lipschitz_on } ((\lambda y. y_{fv} * \omega \text{ } x + \beta \text{ } x) ' U) (\text{the } (\text{activation_tab } N (\varphi \text{ } x)))$

using activation_lipschitz Dense $\langle l \in \text{set } ls \rangle$

by fastforce

moreover have $\text{norm } (\omega \text{ } x) \leq W$

using weight_bounded Dense $\langle l \in \text{set } ls \rangle$ by auto

ultimately show ?thesis

using dense_layer_lipschitz_on[of x W D U N] Dense

by auto

next

case (Activation x)

obtain C where $C\text{-lipschitz_on } U (\text{the } (\text{activation_tab } N (\varphi \text{ } x)))$

using activation_lipschitz Activation $\langle l \in \text{set } ls \rangle$ by fastforce

then show ?thesis

using activation_layer_lipschitz_on Activation

by auto

qed

qed

lemma Rep_fixed_vec_minus: $\text{Rep_fixed_vec } (x - y) = \text{Rep_fixed_vec } x - \text{Rep_fixed_vec } y$

proof -

have $x - y = \text{Abs_fixed_vec } (\text{Rep_fixed_vec } x + \text{smult_vec } (-1) (\text{Rep_fixed_vec } y))$

unfolding minus_fixed_vec_def minus_fixed_vec.rep_eq uminus_fixed_vec.rep_eq

by (simp add: Rep_fixed_vec_inverse)

then have $\text{Rep_fixed_vec } (x - y) = \text{Rep_fixed_vec } x + \text{smult_vec } (-1) (\text{Rep_fixed_vec } y)$

using minus_fixed_vec.rep_eq by blast

also have $\dots = \text{Rep_fixed_vec } x + (- \text{Rep_fixed_vec } y)$

by (simp add: smult_vec_def uminus_vec_def)

also have $\dots = \text{Rep_fixed_vec } x - \text{Rep_fixed_vec } y$

using Rep_fixed_vec by fastforce

finally show ?thesis .

qed

Lipschitz Continuity of Functions (Interval)

Neural Network: Activations lemma *relu_lipschitz'*: $\bigwedge x y. (x::(\text{real}, 'b::\text{finite}) \text{fixed_vec}) \in X \implies$

$(y::(\text{real}, 'b::\text{finite}) \text{fixed_vec}) \in X \implies$

dist

$((\text{Abs_fixed_vec}$
 $(\text{Matrix.vec } (\text{dim_vec } (\text{Rep_fixed_vec } x)))$
 $(\lambda i. \text{if } 0 \leq \text{Rep_fixed_vec } x \$ i \text{ then } \text{Rep_fixed_vec } x \$ i \text{ else } 0)))::(\text{real}, 'b::\text{finite}) \text{fixed_vec})$
 $((\text{Abs_fixed_vec}$
 $(\text{Matrix.vec } (\text{dim_vec } (\text{Rep_fixed_vec } y)))$
 $(\lambda i. \text{if } 0 \leq \text{Rep_fixed_vec } y \$ i \text{ then } \text{Rep_fixed_vec } y \$ i \text{ else } 0)))::(\text{real}, 'b::\text{finite}) \text{fixed_vec})$
 $\leq \text{dist } x y$

proof –

fix $x y::(\text{real}, 'b::\text{finite}) \text{fixed_vec}$ **assume** $x \in X y \in X$

have *dist_eq*: $\text{dist } ((\text{Abs_fixed_vec } (\text{Matrix.vec } (\text{dim_vec } (\text{Rep_fixed_vec } (x::(\text{real}, 'b) \text{fixed_vec})))) (\lambda i. \text{if } 0 \leq \text{Rep_fixed_vec } x \$ i \text{ then } \text{Rep_fixed_vec } x \$ i \text{ else } 0)))::(\text{real}, 'b) \text{fixed_vec})$

$((\text{Abs_fixed_vec } (\text{Matrix.vec } (\text{dim_vec } (\text{Rep_fixed_vec } (y::(\text{real}, 'b) \text{fixed_vec})))) (\lambda i. \text{if } 0 \leq \text{Rep_fixed_vec } y \$ i \text{ then } \text{Rep_fixed_vec } y \$ i \text{ else } 0)))::(\text{real}, 'b) \text{fixed_vec})$

$= \text{norm } ((\text{Abs_fixed_vec } (\text{Matrix.vec } (\text{dim_vec } (\text{Rep_fixed_vec } (x::(\text{real}, 'b) \text{fixed_vec})))) (\lambda i. \text{if } 0 \leq \text{Rep_fixed_vec } x \$ i \text{ then } \text{Rep_fixed_vec } x \$ i \text{ else } 0)))::(\text{real}, 'b) \text{fixed_vec})$

$- (\text{Abs_fixed_vec } (\text{Matrix.vec } (\text{dim_vec } (\text{Rep_fixed_vec } (y::(\text{real}, 'b) \text{fixed_vec})))) (\lambda i. \text{if } 0 \leq \text{Rep_fixed_vec } y \$ i \text{ then } \text{Rep_fixed_vec } y \$ i \text{ else } 0)))::(\text{real}, 'b) \text{fixed_vec})$

by (*simp add: dist_norm*)

have *relu_bound*: $\forall a b::\text{real}. \text{abs } (\max 0 a - \max 0 b) \leq \text{abs } (a - b)$

proof (*intro allI*)

fix $a b :: \text{real}$

have *1-lipschitz_on* $\{a, b\}$ *relu*

using *relu_lipschitz*[*of UNIV*]

by (*rule lipschitz_on_subset*) *auto*

show $\text{abs } (\max 0 a - \max 0 b) \leq \text{abs } (a - b)$

unfolding *lipschitz_on_def relu_def dist_real_def*

by *auto*

qed

have *norm_bound*: $\text{norm } ((\text{Abs_fixed_vec } (\text{Matrix.vec } (\text{dim_vec } (\text{Rep_fixed_vec } x))) (\lambda i. \text{if } 0 \leq \text{Rep_fixed_vec } x \$ i \text{ then } \text{Rep_fixed_vec } x \$ i \text{ else } 0))$

$- \text{Abs_fixed_vec } (\text{Matrix.vec } (\text{dim_vec } (\text{Rep_fixed_vec } y))) (\lambda i. \text{if } 0 \leq \text{Rep_fixed_vec } y \$ i \text{ then } \text{Rep_fixed_vec } y \$ i \text{ else } 0)))::(\text{real}, 'b::\text{finite}) \text{fixed_vec})$

$\leq \text{norm } (x - y)$

proof –

have $\text{norm } ((\text{Abs_fixed_vec } (\text{Matrix.vec } (\text{dim_vec } (\text{Rep_fixed_vec } x))) (\lambda i. \text{if } 0 \leq \text{Rep_fixed_vec } x \$ i \text{ then } \text{Rep_fixed_vec } x \$ i \text{ else } 0))$

$- \text{Abs_fixed_vec } (\text{Matrix.vec } (\text{dim_vec } (\text{Rep_fixed_vec } y))) (\lambda i. \text{if } 0 \leq \text{Rep_fixed_vec } y \$ i \text{ then } \text{Rep_fixed_vec } y \$ i \text{ else } 0)))::(\text{real}, 'b) \text{fixed_vec})$

$= \text{sqrt } (\sum_{i \in \{0..<\text{CARD } ('b)\}}. (\text{norm } (\text{fixed_vec_index } ((\text{Abs_fixed_vec } (\text{Matrix.vec } (\text{dim_vec } (\text{Rep_fixed_vec } x))) (\lambda i. \text{if } 0 \leq \text{Rep_fixed_vec } x \$ i \text{ then } \text{Rep_fixed_vec } x \$ i \text{ else } 0))$

$- \text{Abs_fixed_vec } (\text{Matrix.vec } (\text{dim_vec } (\text{Rep_fixed_vec } y))) (\lambda i. \text{if } 0 \leq \text{Rep_fixed_vec } y \$ i \text{ then } \text{Rep_fixed_vec } y \$ i \text{ else } 0)))::(\text{real}, 'b) \text{fixed_vec } i))^2$

unfolding *norm_fixed_vec_def* **by** *simp*

also have $\dots = \text{sqrt } (\sum_{i \in \{0..<\text{CARD } ('b)\}}. (\text{abs } ((\text{if } 0 \leq \text{vec_index } (\text{Rep_fixed_vec } (x::(\text{real}, 'b) \text{fixed_vec})) i \text{ then } \text{vec_index } (\text{Rep_fixed_vec } (x::(\text{real}, 'b) \text{fixed_vec})) i \text{ else } 0)$

$- (\text{if } 0 \leq \text{vec_index } (\text{Rep_fixed_vec } (y::(\text{real}, 'b) \text{fixed_vec})) i \text{ then } \text{vec_index } (\text{Rep_fixed_vec } (y::(\text{real}, 'b) \text{fixed_vec})) i \text{ else } 0)))^2$

```

proof –
  show ?thesis
  proof –
    have valid_x: Matrix.vec (dim_vec (Rep_fixed_vec x)) (λi. if 0 ≤ Rep_fixed_vec x $ i then Rep_fixed_vec x $ i else 0)
    ∈ carrier_vec CARD('b)
    proof –
      have dim_vec (Rep_fixed_vec x) = CARD('b)
      using Rep_fixed_vec[of x] by simp
      then show ?thesis
      using vec_carrier_vec by blast
    qed
    have valid_y: Matrix.vec (dim_vec (Rep_fixed_vec y)) (λi. if 0 ≤ Rep_fixed_vec y $ i then Rep_fixed_vec y $ i else
    0) ∈ carrier_vec CARD('b)
    proof –
      have dim_vec (Rep_fixed_vec y) = CARD('b)
      using Rep_fixed_vec[of y] by simp
      then show ?thesis using vec_carrier_vec by blast
    qed
    have index_eq: ∀ i < CARD('b).
      fixed_vec_index ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec x)) (λi. if 0 ≤ Rep_fixed_vec x $ i then
      Rep_fixed_vec x $ i else 0)) –
        Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec y)) (λi. if 0 ≤ Rep_fixed_vec y $ i then Rep_fixed_vec
        y $ i else 0)))::(real, 'b)::finite) fixed_vec) i
      = (if 0 ≤ Rep_fixed_vec x $ i then Rep_fixed_vec x $ i else 0) –
        (if 0 ≤ Rep_fixed_vec y $ i then Rep_fixed_vec y $ i else 0)
    proof (clarify)
      fix i assume i < CARD('b)
      have fixed_vec_index ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec x)) (λj. if 0 ≤ Rep_fixed_vec x $ j then
      Rep_fixed_vec x $ j else 0)) –
        Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec y)) (λj. if 0 ≤ Rep_fixed_vec y $ j then Rep_fixed_vec
        y $ j else 0)))::(real, 'b) fixed_vec) i
      = Rep_fixed_vec ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec x)) (λj. if 0 ≤ Rep_fixed_vec x $ j then
      Rep_fixed_vec x $ j else 0)) –
        Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec y)) (λj. if 0 ≤ Rep_fixed_vec y $ j then Rep_fixed_vec
        y $ j else 0)))::(real, 'b) fixed_vec) $ i
      by (simp add: fixed_vec_index_def)
      also have ... = (Rep_fixed_vec ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec x)) (λj. if 0 ≤ Rep_fixed_vec
      x $ j then Rep_fixed_vec x $ j else 0)))::(real, 'b) fixed_vec) –
        Rep_fixed_vec ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec y)) (λj. if 0 ≤ Rep_fixed_vec y $ j
      then Rep_fixed_vec y $ j else 0)))::(real, 'b) fixed_vec)) $ i
      using Rep_fixed_vec_minus
      by (smt (verit, del_insts))
      also have ... = (Matrix.vec (dim_vec (Rep_fixed_vec x)) (λj. if 0 ≤ Rep_fixed_vec x $ j then Rep_fixed_vec x $ j
      else 0) –
        Matrix.vec (dim_vec (Rep_fixed_vec y)) (λj. if 0 ≤ Rep_fixed_vec y $ j then Rep_fixed_vec y $ j else 0)) $ i
      using Abs_fixed_vec_inverse[OF valid_x] Abs_fixed_vec_inverse[OF valid_y] by simp
      also have ... = Matrix.vec (dim_vec (Rep_fixed_vec x)) (λj. if 0 ≤ Rep_fixed_vec x $ j then Rep_fixed_vec x $ j else
      0) $ i –
        Matrix.vec (dim_vec (Rep_fixed_vec y)) (λj. if 0 ≤ Rep_fixed_vec y $ j then Rep_fixed_vec y $ j else 0) $ i
      apply (simp add: minus_vec_def) using ‹(i::nat) < CARD('b)::finite› valid_y by fastforce
      also have ... = (if 0 ≤ Rep_fixed_vec x $ i then Rep_fixed_vec x $ i else 0) –
        (if 0 ≤ Rep_fixed_vec y $ i then Rep_fixed_vec y $ i else 0)
    proof –
      have dim_vec (Rep_fixed_vec x) = CARD('b) using Rep_fixed_vec[of x] by simp

```

```

have dim_vec (Rep_fixed_vec y) = CARD('b) using Rep_fixed_vec[of y] by simp
then show ?thesis using <i < CARD('b)>
  by (simp add: <dim_vec (Rep_fixed_vec (x::(real, 'b)::finite) fixed_vec)) = CARD('b)::finite>)
qed
finally show fixed_vec_index ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec x)) (λi. if 0 ≤ Rep_fixed_vec x
  $ i then Rep_fixed_vec x $ i else 0)) –
  Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec y)) (λi. if 0 ≤ Rep_fixed_vec y $ i then Rep_fixed_vec
  y $ i else 0)))::(real, 'b)::finite) fixed_vec) i
  = (if 0 ≤ Rep_fixed_vec x $ i then Rep_fixed_vec x $ i else 0) –
  (if 0 ≤ Rep_fixed_vec y $ i then Rep_fixed_vec y $ i else 0).
qed
have sqrt (∑ i = 0..<CARD('b). (norm (fixed_vec_index ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec x))
  (λi. if 0 ≤ Rep_fixed_vec x $ i then Rep_fixed_vec x $ i else 0)) –
  Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec y)) (λi. if 0 ≤ Rep_fixed_vec y
  $ i then Rep_fixed_vec y $ i else 0)))::(real, 'b)::finite) fixed_vec) i)^2)
  = sqrt (∑ i = 0..<CARD('b). (norm ((if 0 ≤ Rep_fixed_vec x $ i then Rep_fixed_vec x $ i else 0) –
  (if 0 ≤ Rep_fixed_vec y $ i then Rep_fixed_vec y $ i else 0))^2)
  using index_eq by simp
also have ... = sqrt (∑ i = 0..<CARD('b). abs ((if 0 ≤ Rep_fixed_vec x $ i then Rep_fixed_vec x $ i else 0) –
  (if 0 ≤ Rep_fixed_vec y $ i then Rep_fixed_vec y $ i else 0))^2)
  by (simp add: real_norm_def)
finally show ?thesis.
qed
also have ... ≤ sqrt (∑ i ∈ {0..<CARD('b)}. (abs (Rep_fixed_vec x $ i – Rep_fixed_vec y $ i))^2)
proof –
have ∀ i. abs ((if 0 ≤ Rep_fixed_vec x $ i then Rep_fixed_vec x $ i else 0)
  – (if 0 ≤ Rep_fixed_vec y $ i then Rep_fixed_vec y $ i else 0))
  ≤ abs (Rep_fixed_vec x $ i – Rep_fixed_vec y $ i)
  using relu_bound by (auto simp: max_def)
then show ?thesis
  using sum_mono power_mono
  by (smt (verit, best) real_sqrt_le_mono)
qed
also have ... = norm (x – y)
proof –
have norm (x – y) = sqrt (∑ i ∈ {0..<CARD('b)}. (norm (fixed_vec_index (x – y) i))^2)
  by (simp add: norm_fixed_vec_def)
also have ... = sqrt (∑ i ∈ {0..<CARD('b)}. (norm (Rep_fixed_vec (x – y) $ i))^2)
  by (simp add: fixed_vec_index_def)
also have ... = sqrt (∑ i ∈ {0..<CARD('b)}. (norm ((Rep_fixed_vec x – Rep_fixed_vec y) $ i))^2)
  using Rep_fixed_vec_minus by metis
also have ... = sqrt (∑ i ∈ {0..<CARD('b)}. (norm (Rep_fixed_vec x $ i – Rep_fixed_vec y $ i))^2)
  apply (simp add: minus_vec_def)
  by (metis (no_types, lifting) Finite_Cartesian_Product.sum_cong_aux Rep_fixed_vec atLeastLessThan_iff carrier_vecD
  index_vec)
also have ... = sqrt (∑ i ∈ {0..<CARD('b)}. |Rep_fixed_vec x $ i – Rep_fixed_vec y $ i|^2)
  by simp
finally show ?thesis by simp
qed
finally show norm
  ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec (x::(real, 'b)::finite) fixed_vec))) (λi::nat. if (o::real) ≤
  Rep_fixed_vec x $ i then Rep_fixed_vec x $ i else (o::real))) –
  Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec (y::(real, 'b)::finite) fixed_vec))) (λi::nat. if (o::real) ≤

```

```

Rep_fixed_vec y $ i then Rep_fixed_vec y $ i else (o::real))))::(real, 'b::finite) fixed_vec)
  ≤ norm (x - y) by simp
qed
have ao: map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (λf v. Matrix.vec (dim_vec v) (λi. f (v $ i))) (λb. if o ≤
b then b else o) x =
  Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec x)) (λi. relu (Rep_fixed_vec x $ i)))
  unfolding relu_def max_def
  by (simp add: map_fun_def)
moreover have a1: map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (λf v. Matrix.vec (dim_vec v) (λi. f (v $ i))) (λb.
if o ≤ b then b else o) y =
  Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec y)) (λi. relu (Rep_fixed_vec y $ i)))
  unfolding relu_def max_def
  by (simp add: map_fun_def)
also have a2: dist ((map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (λf v. Matrix.vec (dim_vec v) (λi. f (v $ i)))
(λb. if o ≤ b then b else o) x)::(real, 'b) fixed_vec)
  ((map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (λf v. Matrix.vec (dim_vec v) (λi. f (v $ i))) (λb. if o ≤ b
then b else o) y)::(real, 'b) fixed_vec)
  ≤ 1 * dist x y =
  (dist ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec x)) (λi. if o ≤ Rep_fixed_vec x $ i then Rep_fixed_vec x
$ i else o)))::(real, 'b) fixed_vec)
  ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec y)) (λi. if o ≤ Rep_fixed_vec y $ i then Rep_fixed_vec y $ i
else o)))::(real, 'b) fixed_vec)
  ≤ 1 * dist x y)
  apply (subst ao a1) +
  unfolding relu_def max_def
  by blast
have bo: dist ((map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (λf v. Matrix.vec (dim_vec v) (λi. f (v $ i))) (λb. if
o ≤ b then b else o) x)::(real, 'b) fixed_vec)
  ((map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (λf v. Matrix.vec (dim_vec v) (λi. f (v $ i))) (λb. if o ≤ b
then b else o) y)::(real, 'b) fixed_vec)
  ≤ 1 * dist x y
  using dist_eq norm_bound ao a1 a2 dist_norm
  by (smt (verit, del_insts))
show dist
  ((Abs_fixed_vec
  (Matrix.vec (dim_vec (Rep_fixed_vec x))
  (λi::nat. if o ≤ Rep_fixed_vec x $ i then Rep_fixed_vec x $ i else o)))::(real, 'b) fixed_vec)
  ((Abs_fixed_vec
  (Matrix.vec (dim_vec (Rep_fixed_vec y))
  (λi::nat. if o ≤ Rep_fixed_vec y $ i then Rep_fixed_vec y $ i else o)))::(real, 'b) fixed_vec)
  ≤ dist x y
  using ao a1 a2 bo
  by argo
qed

lemma relu_lipschitz_fv: 1—lipschitz_on (X::(real, 'b::finite) fixed_vec set) (map_fixed_vec relu)
  unfolding lipschitz_on_def relu_def map_fixed_vec_def max_def map_vec_def
  apply (intro conjl impl all)
  subgoal using rel_simps(44) by blast
  subgoal using relu_lipschitz'[of _ X] by simp
done

lemma identity_lipschitz_fv: 1—lipschitz_on (X) (map_fixed_vec identity)

```

```

unfolding lipschitz_on_def relu_def map_fixed_vec_def map_vec_def
by (auto, smt (verit) Rep_fixed_vec_inverse dim_vec eq_vec1 identity_def
  index_vec verit_comp_simplify1(2))

```

```

lemma softplus_lipschitz':  $\bigwedge x y. (x :: (\text{real}, 'b :: \text{finite}) \text{fixed\_vec}) \in X \implies$ 
   $(y :: (\text{real}, 'b :: \text{finite}) \text{fixed\_vec}) \in X \implies$ 
   $\text{dist} ((\text{map\_fun id} (\text{map\_fun Rep\_fixed\_vec Abs\_fixed\_vec}) (\lambda f v. \text{Matrix.vec} (\text{dim\_vec } v) (\lambda i. f (v \$ i)))) \text{softplus}$ 
 $x) :: (\text{real}, 'b) \text{fixed\_vec})$ 
   $(\text{map\_fun id} (\text{map\_fun Rep\_fixed\_vec Abs\_fixed\_vec}) (\lambda f v. \text{Matrix.vec} (\text{dim\_vec } v) (\lambda i. f (v \$ i)))) \text{softplus } y)$ 
   $\leq 1 * \text{dist } x y$ 

```

proof –

```

fix x y :: ( $\text{real}, 'b :: \text{finite}$ )  $\text{fixed\_vec}$  assume  $x \in X y \in X$ 
have softplus_bound:  $\forall a b :: \text{real}. \text{abs} (\text{softplus } a - \text{softplus } b) \leq \text{abs} (a - b)$ 

```

proof (intro allI)

fix a b :: real

have deriv_bound: $\forall (z :: \text{real}) . \text{softplus}' z \leq 1$

proof –

fix z :: real

have softplus' z = $1 / (1 + \exp(-z))$

by (simp add: softplus'_def)

also have ... ≤ 1

by (simp add: add_sign_intros(2))

finally show ?thesis

by (simp add: add_sign_intros(2) softplus'_def)

qed

have cont: continuous_on { $\min a b .. \max a b$ } softplus

by (meson DERIV_isCont softplus' continuous_at_imp_continuous_on)

have diff: $\bigwedge x. \min a b < x \implies x < \max a b \implies \text{softplus}$ differentiable at x

using real_differentiable_def softplus' **by** blast

have mvt: $\exists c. \text{softplus } b - \text{softplus } a = (b - a) * \text{softplus}' c$

proof (cases a = b)

case True

then have $\text{softplus } b - \text{softplus } a = 0$ **by** simp

also have ... = $(b - a) * \text{softplus}' a$ **using** True **by** simp

finally show ?thesis **by** blast

next

case False

obtain l c **where** c_bounds: $\min a b < c \wedge c < \max a b$ **and**

has_deriv: (softplus has_real_derivative l) (at c) **and**

mvt_eq: $\text{softplus} (\max a b) - \text{softplus} (\min a b) = (\max a b - \min a b) * l$

using MVT[of min a b max a b softplus] cont diff False

by fastforce

have l = $\text{softplus}' c$

using has_deriv softplus'[of c]

by (simp add: DERIV_unique)

have result: $\text{softplus } b - \text{softplus } a = (b - a) * \text{softplus}' c$

proof (cases a \leq b)

case True

then have $\min a b = a \max a b = b$ **by** auto

with mvt_eq <l = $\text{softplus}' c$ > **show** ?thesis

by simp

next

```

case False
then have min a b = b max a b = a by auto
with mvt_eq ⟨l = softplus' c⟩ have softplus a - softplus b = (a - b) * softplus' c by simp
then show ?thesis by (simp add: algebra_simps)
qed

```

```

show ?thesis using result by blast
qed

```

```

obtain c where mvt_eq: softplus b - softplus a = (b - a) * softplus' c
using mvt by blast

```

```

have abs (softplus a - softplus b) = abs (softplus b - softplus a)
by simp
also have ... = abs ((b - a) * softplus' c)
using mvt_eq by simp
also have ... = abs (b - a) * abs (softplus' c)
by (simp add: abs_mult)
also have ... ≤ abs (b - a) * 1
using deriv_bound
apply (simp add: mult_left_mono)
by (smt (verit, ccfv_threshold) divide_pos_pos exp_ge_zero mult_left_le softplus'_def)
also have ... = abs (a - b)
by simp
finally show abs (softplus a - softplus b) ≤ abs (a - b) .
qed

```

```

have dist
  ((map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (λ(f::real ⇒ real) v::real Matrix.vec. Matrix.vec (dim_vec v)
  (λi::nat. f (v $ i)))
  softplus x)::(real, 'b::finite) fixed_vec)
  ((map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (λ(f::real ⇒ real) v::real Matrix.vec. Matrix.vec (dim_vec v)
  (λi::nat. f (v $ i)))
  softplus y)::(real, 'b::finite) fixed_vec)
  ≤ 1 * dist x y

```

```

proof -
  have dist_eq: dist ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec (x::(real, 'b) fixed_vec))) (λi. softplus
  (Rep_fixed_vec x $ i))))::(real, 'b) fixed_vec)
  ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec (y::(real, 'b) fixed_vec))) (λi. softplus (Rep_fixed_vec y $
  i))))::(real, 'b) fixed_vec)
  = norm ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec (x::(real, 'b) fixed_vec))) (λi. softplus (Rep_fixed_vec
  x $ i))))::(real, 'b) fixed_vec) -
  (Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec (y::(real, 'b) fixed_vec))) (λi. softplus (Rep_fixed_vec y $
  i))))::(real, 'b) fixed_vec)
  by (simp add: dist_norm softplus_def)

```

```

also have ... ≤ norm (x - y)

```

```

proof -
  have norm ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec x)) (λi. softplus (Rep_fixed_vec x $ i)))
  - Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec y)) (λi. softplus (Rep_fixed_vec y $ i))))::(real, 'b)
  fixed_vec)
  = sqrt (∑ i∈{0..<CARD('b)}. (norm (fixed_vec_index ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec x))
  (λi. softplus (Rep_fixed_vec x $ i)))

```

```

    – Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec y)) (λi. softplus (Rep_fixed_vec
y $ i)))::(real, 'b) fixed_vec i)^2)
  unfolding norm_fixed_vec_def by simp
  also have ... = sqrt (∑ i ∈ {0..<CARD('b)}. (abs (softplus (vec_index (Rep_fixed_vec (x::(real, 'b) fixed_vec) i)
    – softplus (vec_index (Rep_fixed_vec (y::(real, 'b) fixed_vec) i)))^2)
  proof –
  show ?thesis
  proof –
  have valid_x: Matrix.vec (dim_vec (Rep_fixed_vec x)) (λi. softplus (Rep_fixed_vec x $ i)) ∈ carrier_vec CARD('b)
  proof –
  have dim_vec (Rep_fixed_vec x) = CARD('b)
  using Rep_fixed_vec[of x] by simp
  then show ?thesis
  using vec_carrier_vec by blast
  qed
  have valid_y: Matrix.vec (dim_vec (Rep_fixed_vec y)) (λi. softplus (Rep_fixed_vec y $ i)) ∈ carrier_vec CARD('b)
  proof –
  have dim_vec (Rep_fixed_vec y) = CARD('b)
  using Rep_fixed_vec[of y] by simp
  then show ?thesis using vec_carrier_vec by blast
  qed
  have index_eq: ∀ i < CARD('b).
    fixed_vec_index ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec x)) (λi. softplus (Rep_fixed_vec x $ i))) –
      Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec y)) (λi. softplus (Rep_fixed_vec y $ i))))::(real,
'b::finite) fixed_vec) i
    = softplus (Rep_fixed_vec x $ i) – softplus (Rep_fixed_vec y $ i)
  proof (clarify)
  fix i assume i < CARD('b)
  have fixed_vec_index ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec x)) (λj. softplus (Rep_fixed_vec x $
j))) –
    Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec y)) (λj. softplus (Rep_fixed_vec y $ j))))::(real,
'b) fixed_vec) i
    = Rep_fixed_vec ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec x)) (λj. softplus (Rep_fixed_vec x $
j))) –
    Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec y)) (λj. softplus (Rep_fixed_vec y $ j))))::(real,
'b) fixed_vec) $ i
    by (simp add: fixed_vec_index_def)
  also have ... = (Rep_fixed_vec ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec x)) (λj. softplus (Rep_fixed_vec
x $ j))))::(real, 'b) fixed_vec) –
    Rep_fixed_vec ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec y)) (λj. softplus (Rep_fixed_vec y
$ j))))::(real, 'b) fixed_vec)) $ i
    using Rep_fixed_vec_minus
    by (smt (verit, del_insts))
  also have ... = (Matrix.vec (dim_vec (Rep_fixed_vec x)) (λj. softplus (Rep_fixed_vec x $ j)) –
    Matrix.vec (dim_vec (Rep_fixed_vec y)) (λj. softplus (Rep_fixed_vec y $ j))) $ i
    using Abs_fixed_vec_inverse[OF valid_x] Abs_fixed_vec_inverse[OF valid_y] by simp
  also have ... = Matrix.vec (dim_vec (Rep_fixed_vec x)) (λj. softplus (Rep_fixed_vec x $ j)) $ i –
    Matrix.vec (dim_vec (Rep_fixed_vec y)) (λj. softplus (Rep_fixed_vec y $ j)) $ i
    apply (simp add: minus_vec_def) using ‹(i::nat) < CARD('b::finite)› valid_y by fastforce
  also have ... = softplus (Rep_fixed_vec x $ i) – softplus (Rep_fixed_vec y $ i)
  proof –
  have dim_vec (Rep_fixed_vec x) = CARD('b) using Rep_fixed_vec[of x] by simp
  have dim_vec (Rep_fixed_vec y) = CARD('b) using Rep_fixed_vec[of y] by simp
  then show ?thesis using ‹i < CARD('b)›

```

by (simp add: <dim_vec (Rep_fixed_vec (x::(real, 'b::finite) fixed_vec)) = CARD('b::finite)>)
qed
finally show fixed_vec_index ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec x)) (λi. softplus (Rep_fixed_vec x \$ i)))) –
Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec y)) (λi. softplus (Rep_fixed_vec y \$ i))))::(real, 'b::finite) fixed_vec) i
= softplus (Rep_fixed_vec x \$ i) – softplus (Rep_fixed_vec y \$ i) .
qed
have sqrt (∑ i = 0..<CARD('b). (norm (fixed_vec_index ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec x)) (λi. softplus (Rep_fixed_vec x \$ i)))) –
Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec y)) (λi. softplus (Rep_fixed_vec y \$ i))))::(real, 'b::finite) fixed_vec) i)^2)
= sqrt (∑ i = 0..<CARD('b). (norm (softplus (Rep_fixed_vec x \$ i) – softplus (Rep_fixed_vec y \$ i))^2)
using index_eq **by** simp
also have ... = sqrt (∑ i = 0..<CARD('b). abs (softplus (Rep_fixed_vec x \$ i) – softplus (Rep_fixed_vec y \$ i))^2)
by (simp add: real_norm_def)
finally show ?thesis .
qed
qed
also have ... ≤ sqrt (∑ i ∈ {0..<CARD('b)}. (abs (Rep_fixed_vec x \$ i – Rep_fixed_vec y \$ i))^2)
proof –
have ∀ i. abs (softplus (Rep_fixed_vec x \$ i) – softplus (Rep_fixed_vec y \$ i))
≤ abs (Rep_fixed_vec x \$ i – Rep_fixed_vec y \$ i)
using softplus_bound **by** auto
then show ?thesis
using sum_mono power_mono
by (smt (verit, best) real_sqrt_le_mono)
qed
also have ... = norm (x – y)
proof –
have norm (x – y) = sqrt (∑ i ∈ {0..<CARD('b)}. (norm (fixed_vec_index (x – y) i))^2)
by (simp add: norm_fixed_vec_def)
also have ... = sqrt (∑ i ∈ {0..<CARD('b)}. (norm (Rep_fixed_vec (x – y) \$ i))^2)
by (simp add: fixed_vec_index_def)
also have ... = sqrt (∑ i ∈ {0..<CARD('b)}. (norm ((Rep_fixed_vec x – Rep_fixed_vec y) \$ i))^2)
using Rep_fixed_vec_minus **by** metis
also have ... = sqrt (∑ i ∈ {0..<CARD('b)}. (norm (Rep_fixed_vec x \$ i – Rep_fixed_vec y \$ i))^2)
apply (simp add: minus_vec_def)
by (metis (no_types, lifting) Finite_Cartesian_Product.sum_cong_aux Rep_fixed_vec atLeastLessThan_iff carrier_vecD index_vec)
also have ... = sqrt (∑ i ∈ {0..<CARD('b)}. |Rep_fixed_vec x \$ i – Rep_fixed_vec y \$ i|^2)
by simp
finally show ?thesis **by** simp
qed
finally show ?thesis **by** simp
qed
also have ... = dist x y
by (simp add: dist_norm)
have ao: dist ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec x)) (λi. softplus (Rep_fixed_vec x \$ i))))::(real, 'b::finite) fixed_vec) ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec y)) (λi::nat. softplus (Rep_fixed_vec y \$ i))))::(real, 'b::finite) fixed_vec)
≤ dist x y

```

by (metis calculation dist_norm)
  have map_fun_eq_x: map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (λf v. Matrix.vec (dim_vec v) (λi. f (v $ i))) softplus x =
    Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec x)) (λi. softplus (Rep_fixed_vec x $ i)))
  by (simp add: map_fun_def)

  have map_fun_eq_y: map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (λf v. Matrix.vec (dim_vec v) (λi. f (v $ i))) softplus y =
    Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec y)) (λi. softplus (Rep_fixed_vec y $ i)))
  by (simp add: map_fun_def)

  have a1: dist
    ((map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (λ(f::real ⇒ real) v::real Matrix.vec. Matrix.vec (dim_vec v) (λi::nat. f (v $ i))))
    softplus x)::(real, 'b) fixed_vec)
    ((map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (λ(f::real ⇒ real) v::real Matrix.vec. Matrix.vec (dim_vec v) (λi::nat. f (v $ i))))
    softplus y)::(real, 'b) fixed_vec)
    ≤ 1 * dist x y =
    (dist ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec x)) (λi. softplus (Rep_fixed_vec x $ i))))::(real, 'b)::finite)
    fixed_vec) ((Abs_fixed_vec (Matrix.vec (dim_vec (Rep_fixed_vec y)) (λi::nat. softplus (Rep_fixed_vec y $ i))))::(real, 'b)::finite)
    fixed_vec)
    ≤ dist x y
  apply(subst map_fun_eq_x map_fun_eq_y) +
  by simp

  show dist ((map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (λ(f::real ⇒ real) v::real Matrix.vec. Matrix.vec (dim_vec v) (λi::nat. f (v $ i))) softplus x)::(real, 'b)::finite) fixed_vec)
    ((map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (λ(f::real ⇒ real) v::real Matrix.vec. Matrix.vec (dim_vec v) (λi::nat. f (v $ i))) softplus y)::(real, 'b)::finite) fixed_vec)
    ≤ 1 * dist x y using a0 a1 by meson
qed
show
  dist ((map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (λf v. Matrix.vec (dim_vec v) (λi. f (v $ i))) softplus x)::(real, 'b) fixed_vec)
    ((map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (λf v. Matrix.vec (dim_vec v) (λi. f (v $ i))) softplus y)::(real, 'b) fixed_vec)
    ≤ 1 * dist x y
  using
    <dist (map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (λ(f::real ⇒ real) v::real Matrix.vec. Matrix.vec (dim_vec v) (λi::nat. f (v $ i))) softplus (x::(real, 'b)::finite) fixed_vec)) (map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (λ(f::real ⇒ real) v::real Matrix.vec. Matrix.vec (dim_vec v) (λi::nat. f (v $ i))) softplus (y::(real, 'b)::finite) fixed_vec))
    ≤ 1 * dist x y
  by fastforce
qed

lemma softplus_lipschitz: 1—lipschitz_on (X::(real, 'b)::finite) fixed_vec set) (map_fixed_vec softplus)
  unfolding lipschitz_on_def map_fixed_vec_def map_vec_def
  apply(clarsimp)[1]
  using softplus_lipschitz'[of _ X]
  by simp
end

```

5.2 Models

5.2.1 Digraphs as Layers (\exists NN_Digraph_Layers)

theory

NN_Digraph_Layers

imports

NN_Digraph

HOL-Combinatorics.Permutations

begin

definition *layer_equiv* :: ('a list \Rightarrow 'b list) \Rightarrow ('a list \Rightarrow 'b list) \Rightarrow bool (\equiv_l _)

where

\langle layer_equiv f g = (\exists p p'. \forall xs. f xs = permute_list p' (f (permute_list p xs))) \rangle

lemma *mset_eq_layer_equiv*:

assumes \langle mset xs = mset ys \rangle

and \langle mset (f xs) = mset (g ys) \rangle

shows \langle f \equiv_l g \rangle

unfolding *layer_equiv_def* **using** *assms*

by (*metis permute_list_id*)

fun *output_neuron* where

\langle output_neuron (In nid) = False \rangle

| \langle output_neuron (Out nid) = True \rangle

| \langle output_neuron (Neuron n) = False \rangle

fun *input_neuron* where

\langle input_neuron (In nid) = True \rangle

| \langle input_neuron (Out nid) = False \rangle

| \langle input_neuron (Neuron n) = False \rangle

fun *hidden_neuron* where

\langle hidden_neuron (In nid) = False \rangle

| \langle hidden_neuron (Out nid) = False \rangle

| \langle hidden_neuron (Neuron n) = True \rangle

Defining layer types as lists of edges

This subsection details definitions which allow for the easy creation of common layer types. The Activation and Dense layer types map to the layer types used by TensorFlow (see https://www.tensorflow.org/api_docs/python/tf/keras/layers)

Edge construction functions **definition** *mk_edge* :: \langle ('a::one}, 'b, 'c) neural_network \Rightarrow 'a \Rightarrow 'b \Rightarrow 'a \Rightarrow 'a \Rightarrow id \Rightarrow id \Rightarrow ('a, 'b) edge \rangle

where

\langle mk_edge N ω' φ' α' β' id' nid = ($\omega = \omega'$,
tl = (the_elem {n . n \in neurons N \wedge uid n = id'}),
hd = Neuron (λ $\varphi = \varphi'$, $\alpha = \alpha'$, $\beta = \beta'$, uid = nid)) \rangle

definition *mk_out_edge* :: \langle ('a::one}, 'b, 'c) neural_network \Rightarrow id \Rightarrow id \Rightarrow ('a, 'b) edge \rangle

where

```
<mk_out_edge N id' nid' = (| ω = 1,  
  tl = (the_elem {n . n ∈ neurons N ∧ uid n = id'}),  
  hd = Out nid' |)>
```

definition *mk_new_ids* :: ⟨('a::{one}, 'b, 'c) neural_network ⇒ nat list⟩

where

```
<mk_new_ids N = upt (Max(uids (graph N)) + 1)  
  (Max(uids (graph N)) + card (output_layer_ids N) + 1)>
```

mk_new_ids makes a list of new ids corresponding to the size of the current last layer in a given network and the current maximum id in the network. This is used in the activation and out functions in order to generate the new neurons in the edges. In order to help validate that the *mk_new_ids* returns the correct sized list and that the ids are unique in the network the following lemmas are needed to simplify this.

lemma *new_id_len*: ⟨length(*mk_new_ids* N) = length(sorted_list_of_set(output_layer_ids N))⟩

by (simp add: *mk_new_ids_def*)

lemma *new_id_len_card*: ⟨length(*mk_new_ids* N) = card(output_layer_ids N)⟩

by (simp add: *mk_new_ids_def*)

lemma *new_id_distinct*: ⟨distinct(*mk_new_ids* N)⟩

by (metis distinct_upt *mk_new_ids_def*)

lemma *new_id_greater*:

assumes ⟨card (output_layer_ids N) > 0⟩

shows ⟨Min(set(*mk_new_ids* N)) > Max(uids (graph N))⟩

using assms

by (simp add: *mk_new_ids_def*)

lemma *new_id_sorted*:

shows ⟨sorted (*mk_new_ids* N)⟩

by (metis *mk_new_ids_def* sorted_upt)

lemma *new_ids_unique*:

assumes *new_ids_finite*: finite (set(*mk_new_ids* N))

and *current_ids_finite*: finite (uids (graph N))

and *MinMax*: Max (uids (graph N)) < Min (set(*mk_new_ids* N))

shows $uids (graph N) \cap set(mk_new_ids N) = \{\}$

using assms

by (metis *Min_ge* *disjnt_def* *disjnt_ge_max* *disjoint_iff_not_equal* *equalsOD*)

Or by rewriting disjointness:

lemma *new_ids_unique'*:

assumes *new_ids_finite*: finite (set(*mk_new_ids* N))

and *current_ids_finite*: finite (uids (graph N))

and *MinMax*: Max (uids (graph N)) < Min (set(*mk_new_ids* N))

shows $\forall x \in set(mk_new_ids N). x \notin uids (graph N)$

using assms

by (meson *Max_ge* *Min_le* *linorder_not_le* *order_trans*)

Template layer types as list of edges **definition** *dense* :: ⟨('a::{one}, 'b, 'c) neural_network ⇒ nat ⇒ 'a list ⇒ 'b ⇒ 'a ⇒ 'a ⇒ ('a, 'b) edge list⟩

where

```

<dense N n ω' φ' α' β' = (if length ω' = n then
  (let nids = upt (Max(uids (graph N)) + 1) (Max(uids (graph N)) + n + 1)
    in concat(map (λ w . (concat(map
      (λ b . map (λ a . mk_edge N w φ' α' β' a b)
        (sorted_list_of_set(output_layer_ids N))) nids)))) ω'))
  else [])>

```

In *dense* we also take a list of weights which we want our dense layer to be initialised with (requiring another map operator).

```

definition out :: (<'a::{one}, 'b, 'c> neural_network ⇒ ('a, 'b) edge list>
where
<out N = (let nids = mk_new_ids N;
  nedges = map (λ a . mk_out_edge N (fst a) (snd a))
    (zip (sorted_list_of_set(output_layer_ids N)) nids)
  in (if distinct nedges then nedges else []))>

```

```

definition activation :: (<'a::{one}, 'b, 'c> neural_network ⇒ 'b ⇒ 'a ⇒ 'a ⇒ ('a, 'b) edge list>
where
<activation N φ' α' β' = (let nids = mk_new_ids N;
  nedges = map (λ a . mk_edge N 1 φ' α' β' (fst a) (snd a))
    (zip (sorted_list_of_set(output_layer_ids N)) nids)
  in (if distinct nedges then nedges else []))>

```

here we call *mk_edge* with the weight ω set to 1 as we do not want to change the output of the previous layer (we are simply applying the activation function)

```

definition <add_edges N edge_list = foldr (λ a b. add_nn_edge b a) edge_list (graph N)>
definition <add_out N = add_edges N (out N) >
definition <add_dense N n ω' φ' α' β' = add_edges N (dense N n ω' φ' α' β')>
definition <add_activation N φ' α' β' = add_edges N (activation N φ' α' β')>

```

definitions *add_edges*, *add_out*, *add_dense* and *add_activation* allow for easy addition of TensorFlow layer types to an existing Neural Network.

Defining Layers in the Digraph Model

```

fun layersdigraph :: <nat ⇒ ('a::{zero, linorder, numeral}, 'b, 'c) neural_network
  ⇒ ('a, 'b) edge ⇒ ('a × error)>
where
<layersdigraph _ N ((ω=_, tl=_, hd=In _)) = (o, ERROR)>
| <layersdigraph _ N ((ω=_, tl=Out _ , hd=_)) = (o, ERROR)>
| <layersdigraph _ N ((ω=_, tl=In uidin , hd=_)) = (o, OK)>
| <layersdigraph n N e = (if 0 < n then
  (let
    tl' = (case (tl e) of (Neuron t) ⇒ t);
    E' = incoming_arcs N (Neuron.uid tl');
    lvals = ((λ e'. (case layersdigraph (n-1) N e' of
      (_, ERROR) ⇒ ((o,o), ERROR)
      | (v, OK) ⇒ ((v+1, uid (tl e')), OK))) 'E')
  in
    (Max ((λ a . fst (fst a)) ' {n. n ∈ lvals ∧ snd n = OK } ), OK))
  else (o, ERROR))>

```

Layers are defined as the path from the output node, this allows all dependencies to be calculated before prediction. In *layers_{digraph}* the layer is calculated using the edges.

```

fun layersdigraph_neuron :: <nat ⇒ ('a::{zero,linorder,numeral}, 'b, 'c) neural_network
  ⇒ ('a, 'b) neuron ⇒ ('a × error)>
where
  <layersdigraph_neuron _ N (In uidin) = (o, OK)>
  | <layersdigraph_neuron n N (Out uidout) = (if o < n then
    (let
      E' = tl ' (incoming_arcs N uidout);
      lvals = ((λ n'. (case layersdigraph_neuron (n-1) N n' of
        (_, ERROR) ⇒ ((o,o), ERROR)
        | (v, OK) ⇒ ((v+1, uid n'), OK))) ' E')
      in (Max ((λ a .fst(fst a)) ' {n. n ∈ lvals ∧ snd n = OK}), OK))
    else (o, ERROR))>
  | <layersdigraph_neuron n N (Neuron a) = (if o < n then
    (let
      E' = tl ' (incoming_arcs N (Neuron.uid a));
      lvals = ((λ n'. (case layersdigraph_neuron (n-1) N n' of
        (_, ERROR) ⇒ ((o,o), ERROR)
        | (v, OK) ⇒ ((v+1, uid n'), OK))) ' E')
      in (Max ((λ a .fst(fst a)) ' {n. n ∈ lvals ∧ snd n = OK}), OK))
    else (o, ERROR))>

```

In `layersdigraph_neuron` the layer is calculated using the neurons instead, this is more intuitive as it is the neurons that are arranged in layers.

Defining the behaviour of layers `fun layersedges :: 'a ⇒ 'a ⇒ ('a::{zero,numeral,linorder}, 'b, 'c) neural_network`

```

  ⇒ ('a, 'b) edge set> where
  <layersedges l l' N = (let nall = neurons N;
    layer = (λ n . ((layersdigraph_neuron (card nall) N n), uid n)) ' nall;
    nin = snd ' {n . n ∈ layer ∧ fst(fst n) = l};
    nout = snd ' {n . n ∈ layer ∧ fst(fst n) = l'}
    in { e . e ∈ edges N ∧ uid (tl e) ∈ nin ∧ uid (hd e) ∈ nout } )>

```

get all edges between layer n and n+1

Predicates to distinguish different layer types The following for functions test whether sets of edges correspond to the correct type of connections for Dense, Activation, Input and Output layers.

definition `isDenses :: ('a, 'b) edge set ⇒ bool` **where**
`<isDenses e = ((∀ n' ∈ tl ' e . ∀ n'' ∈ hd ' e . ∃ e' ∈ e . tl e' = n' ∧ hd e' = n''))>`

definition `isActivations :: ('a, 'b) edge set ⇒ bool` **where**
`<isActivations e = ((∀ n' ∈ tl ' e . ∃! e' ∈ e . tl e' = n') ∧ (∀ n'' ∈ hd ' e . ∃! e'' ∈ e . hd e'' = n''))>`

definition `isInputs :: ('a, 'b) edge set ⇒ bool` **where**
`<isInputs e = (isDenses e ∧ (∀ n ∈ hd ' e . input_neuron n))>`

definition `isOutputs :: ('a, 'b) edge set ⇒ bool` **where**
`<isOutputs e = (isActivations e ∧ (∀ n''' ∈ hd ' e . output_neuron n'''))>`

The following for functions test whether lists of edges correspond to the correct type of connections for Dense, Activation, Input and Output layers. We want these definitions over lists and sets in order to allow us to use whichever is more efficient in specific situations.

definition $isDense_l :: \langle 'a, 'b \rangle \text{ edge list} \Rightarrow \text{bool}$ **where**
 $\langle isDense_l e = (\text{let } t = (\text{map } tl \ e); h = (\text{map } hd \ e) \text{ in}$
 $(\forall n' \in \text{set } t . \forall n'' \in \text{set } h .$
 $\text{filter } (\lambda e' . tl \ e' = n' \wedge hd \ e' = n'') \ e \neq []) \rangle$

definition $isInput_l :: \langle 'a, 'b \rangle \text{ edge list} \Rightarrow \text{bool}$ **where**
 $\langle isInput_l e = (isDense_l \ e \wedge \text{foldr } (\wedge) \ (\text{map } input_neuron \ (\text{map } hd \ e)) \ \text{True}) \rangle$

definition $isActivation_l :: \langle 'a, 'b \rangle \text{ edge list} \Rightarrow \text{bool}$ **where**
 $\langle isActivation_l e = (\text{let } t = (\text{map } tl \ e); h = (\text{map } hd \ e) \text{ in}$
 $\text{distinct } t \wedge \text{distinct } h \wedge \text{length } t = \text{length } h \wedge \text{length } e = \text{length } h \wedge$
 $\text{length } t = \text{length } e) \rangle$

definition $isOutput_l :: \langle 'a, 'b \rangle \text{ edge list} \Rightarrow \text{bool}$ **where**
 $\langle isOutput_l e = (isActivation_l \ e \wedge \text{foldr } (\wedge) \ (\text{map } (\text{output_neuron} \circ \text{hd}) \ e) \ \text{True}) \rangle$

Prove that the list and set definitions of our layers define the same behaviour, e.g. it does not matter whether $isActivation_l$ or $isActivation_s$ is used, the same connections are ensured

lemma $allOutput$:

shows $\langle \text{foldr } (\wedge) \ (\text{map } (\text{output_neuron} \circ \text{hd}) \ e) \ \text{True} = (\forall n' \in \text{hd}' \ \text{set } e . \text{output_neuron } n') \rangle$

proof ($induction \ e$)

case Nil

then show $?case \ \text{by } simp$

next

case ($Cons \ a \ e$)

then show $?case \ \text{by } simp$

qed

lemma $allInput$:

shows $\langle \text{foldr } (\wedge) \ (\text{map } (\text{input_neuron} \circ \text{hd}) \ e) \ \text{True} = (\forall n' \in \text{hd}' \ \text{set } e . \text{input_neuron } n') \rangle$

proof ($induction \ e$)

case Nil

then show $?case \ \text{by } simp$

next

case ($Cons \ a \ e$)

then show $?case \ \text{by } simp$

qed

lemma $forAll$:

$\langle (\forall n' \in \text{set } (\text{map } tl \ e) . \forall n'' \in \text{set } (\text{map } hd \ e) . \text{filter } (\lambda e' . tl \ e' = n' \wedge hd \ e' = n'') \ e \neq []) =$
 $(\forall n' \in \text{tl}' \ \text{set } e . \forall n'' \in \text{hd}' \ \text{set } e . \exists e' \in \text{set } e . tl \ e' = n' \wedge hd \ e' = n'') \rangle$

proof ($induction \ e$)

case Nil

then show $?case \ \text{by } simp$

next

case ($Cons \ a \ e$)

then show $?case \ \text{by } (\text{smt } (\text{verit}, \text{del_insts}) \ \text{empty_filter_conv } \text{list.set_map})$

qed

lemma $isDense_l_isDense_s_equivalence$: $\langle isDense_l \ E = isDense_s \ (\text{set } E) \rangle$

apply ($safe$)

subgoal **apply** ($simp \ \text{add}: isDense_l_def \ isDense_s_def$)

subgoal **apply** ($\text{metis } (\text{mono_tags}, \text{lifting}) \ \text{empty_filter_conv}$) **done**

```

subgoal apply (simp add: isDensel_def isDenses_def)
  subgoal apply (smt (verit, ccfv_threshold) empty_filter_conv) done
done

```

```

lemma isInputl_isInputs_equivalence: <isInputl E = isInputs (set E)>
apply(safe)
subgoal apply (simp add: isInputl_def isInputs_def)
  using allInput isDensel_isDenses_equivalence by auto
subgoal apply (simp add: isInputl_def isInputs_def)
  using allInput isDensel_isDenses_equivalence by auto
done

```

```

lemma isActivationl_isActivations_equivalence:
assumes distinct: <distinct E>
shows <isActivationl E = isActivations (set E)>
using assms
apply(safe)
subgoal apply(simp add: isActivationl_def isActivations_def)
  by (metis distinct_map inj_onD)
subgoal apply(simp add: isActivationl_def isActivations_def)
  using distinct_map inj_on_def by fastforce
done

```

```

lemma isOutputl_isOutputs_equivalence:
assumes distinct: <distinct E>
shows <isOutputl E = isOutputs (set E)>
using assms
apply (safe)
subgoal
  apply (simp add: isOutputl_def isOutputs_def)
  using allOutput isActivationl_isActivations_equivalence by blast
subgoal
  apply (simp add: isOutputl_def isOutputs_def)
  using allOutput isActivationl_isActivations_equivalence by blast
done

```

We currently support the following 4 types of layers:

```

definition <layersinput l l' N = isInputs (layersedges l l' N)>
definition <layersoutput l l' N = isOutputs (layersedges l l' N)>
definition <layersdense l l' N = isDenses (layersedges l l' N)>
definition <layersactivation l l' N = isActivations (layersedges l l' N)>

```

Conversion of layer types

The following helper lemmas are needed to prove that tails are unique within the edge lists. context `neural_network_digraph` begin

```

lemma nn_pregraph (graph N)
by (meson neural_network_digraph_axioms neural_network_digraph_def nn_graph.axioms(1))

```

```

lemma uid_is_singleton: <x ∈ NN_Digraph.uid ' (neurons N)
  ⇒ is_singleton {n ∈ neurons N. NN_Digraph.uid n = x}>
using neurons_def o_def nn_pregraph.id_vert_inj

```

```
by (smt (verit, best) empty_iff image_iff inj_onD is_singleton!
    mem_Collect_eq neural_network_digraph_axioms neural_network_digraph_def nn_graph.id_vert_inj)
```

lemma distinct_elem:

```
assumes a1: <distinct X >
and a2: <set X ⊆ uid ' (neurons N) >
shows <distinct (map (λx. the_elem {n ∈ neurons N. NN_Digraph.uid n = x}) X)>
by (smt (verit, best) a1 a2 distinct_map image_iff inj_on_def is_singleton_the_elem
    mem_Collect_eq subset_code(1) uid_is_singleton)
```

lemma output_layer_ids_subset_neuron_ids: <output_layer_ids N ⊆ uid ' (neurons N) >

```
unfolding image_def neurons_def output_layer_ids_def o_def output_layer_def output_verts_def
by auto
```

end

Activation layer proofs **lemma distinct_activation_edges:** <distinct (activation N φ' α' β')>

```
apply (simp add: activation_def)
by (smt (verit) distinct.simps(1))
```

lemma output_activation_layer_length_equal:

```
assumes notEmptyNeurons: <neurons N ≠ {}>
and notEmptyActivationLayer: <length(activation N  $\varphi'$   $\alpha'$   $\beta'$ ) ≠ 0>
shows <card(output_layer_ids N) = length(activation N  $\varphi'$   $\alpha'$   $\beta'$ )>
using assms
apply (simp add: activation_def mk_new_ids_def mk_out_edge_def output_layer_ids_def)
apply auto
done
```

lemma new_ids_activation_layer_length_equal:

```
assumes notEmptyNeurons: <neurons N ≠ {}>
and notEmptyActivationLayer: <length(activation N  $\varphi'$   $\alpha'$   $\beta'$ ) ≠ 0>
and notEmptyNewIds: <length(mk_new_ids N) ≠ 0>
shows <length(mk_new_ids N) = length(activation N  $\varphi'$   $\alpha'$   $\beta'$ )>
using assms
apply (simp add: out_def mk_new_ids_def)
apply (metis assms(2) output_activation_layer_length_equal)
done
```

lemma map_neuron_hd_id:

```
<(map (λx. Neuron (|φ = φ', α = α', β = β', uid = f x|)) X) =
  (map (λx. Neuron (|φ = φ', α = α', β = β', uid = x|)) (map f X))>
by simp
```

lemma map_neuron_tl_id:

```
<(map (λx. the_elem {n ∈ neurons N. NN_Digraph.uid n = f x}) X) =
  (map (λx. the_elem {n ∈ neurons N. NN_Digraph.uid n = x})(map f X))>
by simp
```

context nn_pregraph begin

lemma distinct_head_activation: <distinct(map hd (activation N φ' α' β'))>

```
apply (simp add: activation_def Let_def mk_edge_def o_def)
```

```

apply (simp only: map_neuron_hd_id[of _ _ _ snd _] map_snd_zip new_id_len)
using new_id_distinct
apply (simp add: distinct_conv_nth)
by auto

```

end

context neural_network_digraph **begin**

```

lemma distinct_tail_activation: <distinct(map tl (activation N  $\varphi'$   $\alpha'$   $\beta'$ ))>
apply (simp add: activation_def Let_def mk_edge_def o_def)
apply (simp only: map_neuron_tl_id[of _ <fst>] map_fst_zip new_id_len)
using distinct_elem[of <sorted_list_of_set (output_layer_ids N)>]
      distinct_sorted_list_of_set[of <output_layer_ids N>]
      output_layer_ids_subset_neuron_ids set_sorted_list_of_set
by (metis bot.extremum set_empty sorted_list_of_set.fold_insort_key.infinite)

```

```

lemma activation_is_activation: <isActivationl (activation N  $\varphi'$   $\alpha'$   $\beta'$ )>
apply (simp add: isActivationl_def)
apply (safe)
subgoal apply (rule distinct_tail_activation) done
subgoal apply (rule nn_pregraph.distinct_head_activation)
      apply (rule NN_Digraph.nn_pregraph_mk) done
done

```

end

Output layer proofs **lemma** output_output_layer_length_equal:

```

assumes notEmptyNeurons: <neurons N  $\neq$  {}>
and notEmptyOutputLayer: <length(out N)  $\neq$  0>
shows <card(output_layer_ids N) = length(out N)>
using assms
apply (simp add: out_def mk_new_ids_def mk_out_edge_def output_layer_ids_def)
by auto

```

```

lemma new_ids_output_layer_length_equal:
assumes notEmptyNeurons: <neurons N  $\neq$  {}>
and notEmptyOutputLayer: <length(out N)  $\neq$  0>
shows <length(mk_new_ids N) = length(out N)>
using assms
apply (simp add: out_def)
using output_output_layer_length_equal new_id_len
apply (simp add: new_id_len notEmptyNeurons out_def output_output_layer_length_equal)
done

```

```

lemma distinct_output_edges: <distinct (out N)>
apply (smt (verit, best) out_def card.empty card_distinct list.set(1) list.size(3))
done

```

```

lemma map_out_neuron_hd_id: <(map ( $\lambda x$ . Out (f x)) X) = (map ( $\lambda x$ . Out x) (map f X))>
by simp

```

context nn_pregraph **begin**

```

lemma distinct_head_output: <distinct(map hd (out N))>
  apply (simp add: out_def Let_def mk_out_edge_def o_def)
  apply (simp only: map_out_neuron_hd_id[of snd _] map_snd_zip new_id_len)
  using new_id_distinct
  apply (simp add: distinct_conv_nth)
  by auto

```

end

```

lemma fold_and_map: <foldr (∧) (map (λx. True) X) True>
proof (induction X)

```

```

  case Nil
  then show ?case by simp

```

next

```

  case (Cons a X)
  then show ?case by simp

```

qed

```

lemma head_output_neurons: <foldr (∧) (map (output_neuron ∘ edge.hd) (out N)) True>

```

```

  apply (simp add: o_def out_def Let_def mk_out_edge_def)
  using fold_and_map
  by(auto)

```

context neural_network_digraph **begin**

```

lemma distinct_tail_output: <distinct(map tl (out N))>

```

```

  apply (simp add: out_def Let_def mk_out_edge_def o_def)
  apply (simp only: map_neuron_tl_id[of _ fst _] map_fst_zip new_id_len)
  using distinct_sorted_list_of_set distinct_elem
  by (metis (mono_tags, lifting) distinct.simps(1) list.simps(8) output_layer_ids_subset_neuron_ids
    sorted_list_of_set.fold_insort_key.infinite sorted_list_of_set.set_sorted_key_list_of_set)

```

```

lemma output_is_output: <isOutputl (out N)>

```

```

  apply (simp add: isOutputl_def isActivationl_def)
  apply (safe)
  subgoal apply (rule distinct_tail_output) done
  subgoal apply (rule nn_pregraph.distinct_head_output) apply(rule NN_Digraph.nn_pregraph_empty) done
  subgoal apply (rule head_output_neurons) done
  done

```

Dense layer proofs **lemma** dense_is_dense:

```

  assumes neuronsNotZero: <n > 0>
  and weightEqualNeurons: <length ω' = n>
  shows <isDenses(set(dense N n ω' φ' α' β'))>
  using assms
  apply (safe)
  apply (simp add: isDenses_def dense_def activation_def)
  apply (simp add: neurons_def output_layer_ids_def output_layer_def output_verts_def mk_edge_def)
  apply (force)
  done

```

end

end

5.2.2 Neural Network as Sequential Layers using Lists (\exists NN_Layers_List_Main)

theory

 NN_Layers_List_Main

imports

 Main

 NN_Layers

 HOL-Library.Interval

 Properties

begin

definition $\langle \text{valid_activation_tab}_l \text{ tab} = (\forall f \in \text{ran tab. } \forall \text{ xs. length xs} = \text{length (f xs)}) \rangle$

lemma *valid_activation_preserves_length*:

assumes $\langle \text{valid_activation_tab}_l \text{ t} \rangle$

assumes $\langle t \ n = \text{Some } f \rangle$

shows $\langle \text{length xs} = \text{length (f xs)} \rangle$

using *assms unfolding valid_activation_tab_l_def*

by (*simp add: ranI*)

fun *layer_consistent_l* :: $('a \text{ list, } 'b, 'a \text{ list list}) \text{ neural_network_seq_layers} \Rightarrow \text{nat} \Rightarrow ('a \text{ list, } 'b, 'a \text{ list list}) \text{ layer} \Rightarrow \text{bool}$

where

$\langle \text{layer_consistent}_l \text{ _ nc (In } l) = (o < \text{units } l \wedge \text{nc} = \text{units } l) \rangle$

$\langle \text{layer_consistent}_l \text{ _ nc (Out } l) = (o < \text{units } l \wedge \text{nc} = \text{units } l) \rangle$

$\langle \text{layer_consistent}_l \text{ N nc (Activation } l) = ((o < \text{units } l \wedge \text{nc} = \text{units } l) \wedge (((\text{activation_tab } N) (\varphi \ l)) \neq \text{None})) \rangle$

$\langle \text{layer_consistent}_l \text{ N nc (Dense } l) = (o < \text{units } l \wedge o < \text{nc} \wedge \text{length } (\beta \ l) = \text{units } l \wedge \text{length } (\omega \ l) = \text{units } l \wedge (\forall r \in \text{set } (\omega \ l). \text{length } r = \text{nc}) \wedge (((\text{activation_tab } N) (\varphi \ l)) \neq \text{None})) \rangle$

fun *layers_consistent_l* **where**

$\langle \text{layers_consistent}_l \text{ N } [] = \text{True} \rangle$

$\langle \text{layers_consistent}_l \text{ N w (l\#ls)} = ((\text{layer_consistent}_l \text{ N w } l) \wedge (\text{layers_consistent}_l \text{ N (out_deg_layer } l) \ \text{ls})) \rangle$

lemma *layer_consistent_l_in_deg_layer*:

assumes *layer_consistent_l* N nc l

shows *in_deg_layer* l = nc

proof(*cases* l)

case (In x1)

then show ?*thesis* **using** *assms by simp*

next

case (Out x2)

then show ?*thesis* **using** *assms by simp*

next

case (Dense x3)

then show ?*thesis* **using** *assms by simp*

next

case (Activation x4)

then show ?*thesis* **using** *assms by simp*

qed

lemma *layers_consistent_l_in_deg*:

```

  assumes (layers_consistentl N nc (l#ls'))
shows in_deg_layer l = nc
proof(insert assms, cases l)
  case (In x1)
  then show ?thesis
    using assms by simp
next
  case (Out x2)
  then show ?thesis using assms by simp
next
  case (Dense x3)
  then show ?thesis using assms by simp
next
  case (Activation x4)
  then show ?thesis using assms by simp
qed

```

lemma *layer_consistent_l_activation_tab_const*:
 $\langle \text{layer_consistent}_l N nc l = \text{layer_consistent}_l (\{ \text{layers} = \text{ls}, \text{activation_tab} = \text{activation_tab } N \}) nc l \rangle$
 by(cases l, simp_all)

lemma *layers_consistent_l_activation_tab_const*:
 $\langle \text{layers_consistent}_l N nc \text{ls} = \text{layers_consistent}_l (\{ \text{layers} = \text{ls}', \text{activation_tab} = \text{activation_tab } N \}) nc \text{ls} \rangle$
 proof(induction ls arbitrary: nc)
 case Nil
 then show ?case by simp
 next
 case (Cons a ls)
 then show ?case
 by(simp add: layer_consistent_l_activation_tab_const[symmetric])
 qed

lemma *layers_consistent_l_layersN_const*:
 $\langle \text{layers_consistent}_l N = \text{layers_consistent}_l (\{ \text{layers} = \text{ls}', \text{activation_tab} = \text{activation_tab } N \}) \rangle$
 using layers_consistent_l_activation_tab_const by blast

lemma *layers_consistent_lAll*:
 assumes $\langle \text{layers_consistent}_l N \text{inputs} (\text{layers } N) \rangle$
 shows $\langle \forall l \in \text{set } (\text{layers } N). \exists n . \text{layer_consistent}_l N n l \rangle$
 proof(cases N)
 case (fields layers activation_tab)
 then have $\forall l \in \text{set } \text{layers}. \exists n . \text{layer_consistent}_l (\{ \text{layers} = \text{layers}, \text{activation_tab} = \text{activation_tab} \}) n l$
 proof(insert assms[simplified fields, simplified], induction layers arbitrary:inputs activation_tab N)
 case Nil
 then show ?case by simp
 next
 case (Cons a layers)
 then show ?case
 apply(simp)
 using Cons fields layers_consistent_l_activation_tab_const layer_consistent_l_activation_tab_const
 neural_network_seq_layers.select_convs(2)
 by metis

```

qed
then
show ?thesis
  using fields by simp
qed

```

```

lemma layers_consistentlAll':
  assumes <layers_consistentl N (in_deg_NN N) (layers N)>
  shows <∀ l ∈ set (layers N). ∃ n . layer_consistentl N n b>
  using assms layers_consistentlAll' by blast

```

```

lemma layers_consistentl_layer_consistentl_Dense:
  assumes <layers_consistentl N (in_deg_NN N) (layers N)>
  and (Dense x3) ∈ set (layers N)
  shows <layer_consistentl N (length (ω x3 ! o)) (Dense x3) >
  using assms layer_consistentl.simps(4)[of N (length (ω x3 ! o)) x3]
  by (metis layer_consistentl.simps(4) layers_consistentlAll' nth_mem)

```

```

lemma layers_consistentl_layer_consistentl_Activation:
  assumes <layers_consistentl N (in_deg_NN N) (layers N)>
  and (Activation x3) ∈ set (layers N)
  shows <layer_consistentl N (units x3) (Activation x3) >
  using assms layer_consistentl.simps(3)[of N _ x3]
  by (meson layer_consistentl.simps(3) layers_consistentlAll)

```

```

locale neural_network_sequential_layersl =
  fixes N::<(a::comm_ring list, 'b, 'a list list) neural_network_seq_layers>
  assumes head_is_In: <isIn (hd (layers N))>
  and last_is_Out: <isOut (last (layers N))>
  and layer_internal: <list_all isInternal ((tl o butlast) (layers N))>
  and activation_tab_valid: <valid_activation_tabl (activation_tab N)>
  and layer_valid: <layers_consistentl N (in_deg_NN N) (layers N)>
begin
lemma layers_nonempty: <layers N ≠ []>
  by (metis hd_Nil_eq_last head_is_In isIn.elims(2) isOut.simps(2) last_is_Out)

```

```

lemma min_length_layers_two: <1 < length (layers N)>
  by (metis (no_types, lifting) One_nat_def add_o append_Nil append_butlast_last_id
  append_eq_append_conv head_is_In isIn.simps(2) isOut.elims(2) last_is_Out
  layers_nonempty length_o_conv less_one linorder_neqE_nat list.sel(1) list.size(4))

```

```

lemma layers_structure: <∃ il ol ls. layers N = (In il)#ls@[Out ol]>
  by (metis (no_types, lifting) append_butlast_last_id head_is_In isIn.elims(2) isOut.elims(2)
  last.simps last_is_Out layer.distinct(1) layers_nonempty list.exhaust list.sel(1))

```

```

end

```

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_l* to describe the common concepts of all neural network models that use layers as 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

fun $\text{predict}_{\text{layer}_l} :: \langle 'a :: \{\text{monoid_add}, \text{times}\} \text{ list}, 'b, 'a \text{ list list} \rangle \text{ neural_network_seq_layers} \Rightarrow ('a \text{ list}) \text{ option} \Rightarrow ('a \text{ list}, 'b, 'a \text{ list list}) \text{ layer} \Rightarrow ('a \text{ list}) \text{ option}$

where

```

  <predictlayer_l N (Some vs) (In l) = (if layer_consistentl N (length vs) (In l) then Some vs else None)>
  | <predictlayer_l N (Some vs) (Out l) = (if layer_consistentl N (length vs) (Out l) then Some vs else None)>
  | <predictlayer_l N (Some vs) (Dense pl) = (if layer_consistentl N (length vs) (Dense pl) then
    (let
      in_w_pairs = map (λ e. zip vs e) (ω pl);
      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 )))
    else None)
  >
  | <predictlayer_l N (Some vs) (Activation pl) = (if layer_consistentl N (length vs) (Activation pl) then
    (case activation_tab N (φ pl) of
      None ⇒ None
    | Some f ⇒ Some (f vs))
    else None)
  >
  | <predictlayer_l _ None _ = None>

```

lemma $\text{length_out}: \langle \text{predict}_{\text{layer}_l} N' \text{ vs} \rangle (\text{Out } l) = \text{Some } \text{res} \implies \text{length}(\text{res}) = (\text{units } l)$
by(cases vs, auto split:if_splits)

fun

$\text{predict}_{\text{layer}_l_impl} :: \langle 'a :: \{\text{monoid_add}, \text{times}\} \text{ list}, 'b, 'a \text{ list list} \rangle \text{ neural_network_seq_layers} \Rightarrow 'a \text{ list} \Rightarrow ('a \text{ list}, 'b, 'a \text{ list list}) \text{ layer} \Rightarrow 'a \text{ list}$

where

```

  <predictlayer_l_impl N vs (In l) = vs>
  | <predictlayer_l_impl N vs (Out l) = vs>
  | <predictlayer_l_impl N vs (Dense pl) = (let
    in_w_pairs = map (λ e. zip vs e) (ω pl);
    wsums     = map (λ vs'. ∑ (x,y)←vs'. x*y) in_w_pairs;
    wsum_bias = map (λ (s,b). s+b) (zip wsums (β pl));
    φl = the (activation_tab N (φ pl))
  in
    φl wsum_bias)
  >
  | <predictlayer_l_impl N vs (Activation pl) = (let
    φl = the (activation_tab N (φ pl))
  in
    φl vs)
  >

```

definition $\langle \text{predict}_{\text{seq_layer}_l} N \text{ inputs} = \text{foldl} (\text{predict}_{\text{layer}_l} N) (\text{Some } \text{inputs}) (\text{layers } N) \rangle$

definition $\langle \text{predict}_{\text{seq_layer}_l_impl} N \text{ inputs} = \text{foldl} (\text{predict}_{\text{layer}_l_impl} N) \text{ inputs} (\text{layers } N) \rangle$

lemma $\text{predict_layer_Some}$:

assumes $\langle (\text{layer_consistent}_l N (\text{length } \text{xs}) l) \rangle$

shows $\langle (\text{predict}_{\text{layer}_l} N (\text{Some } \text{xs}) l \neq \text{None}) \rangle$

```

proof(cases l)
  case (In x1)
  then show ?thesis using assms by(simp)
next
  case (Out x2)
  then show ?thesis using assms by simp
next
  case (Dense x3)
  then show ?thesis using assms by force
next
  case (Activation x4)
  then show ?thesis using assms by force
qed

```

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 $predict_{layer_l}$ function. The dense layer of the network is where the weighted sum is calculated, case three in $predict_{layer_l}$, 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 $predict_{layer_l}$ over the network from left to right ($foldl$) in $predict_{layer_l}$.

```

lemma fold_predict_L_strict: <(foldl (predict_{layer\_l} N) None ls) = None>
  by(induction ls, simp_all)

```

```

lemmas [nn_layer] = predict_{layer\_l}.simps predict_layer_Some fold_predict_L_strict

```

```

lemma predict_{layer\_l}_activation_tab: assumes activation_tab N = activation_tab N' shows

```

```

  <predict_{layer\_l} N x xs = predict_{layer\_l} N' x xs>

```

```

proof(cases xs)
  case (In x1)
  then show ?thesis proof(cases x)
    case None
    then show ?thesis using In assms by(simp)
  next
    case (Some a)
    then show ?thesis using In assms by(simp)
  qed
next
  case (Out x2)
  then show ?thesis proof(cases x)
    case None
    then show ?thesis using Out assms by(simp)
  next
    case (Some a)
    then show ?thesis using Out assms by(simp)
  qed
next
  case (Dense x3)
  then show ?thesis proof(cases x)
    case None
    then show ?thesis by simp
  next

```

```

  case (Some a)
  then show ?thesis by(auto simp add:assms Dense)
qed
next
case (Activation x4)
then show ?thesis proof(cases x)
  case None
  then show ?thesis using Activation assms by(simp)
next
case (Some a)
then show ?thesis using Activation assms by(simp)
qed
qed

```

```

lemma predictlayer_l_activation_tab_const: ⟨predictlayer_l N = predictlayer_l (⟦layers = l, activation_tab = activation_tab N⟧)⟩
  apply(rule ext)+
  by (metis neural_network_seq_layers.select_convs(2) predictlayer_l_activation_tab)

```

```

lemma input_layer:
  assumes ⟨y = length i⟩ and ⟨0 < y⟩
  shows ⟨predictlayer_l N (Some i) (In (⟦name = x, units = y⟧)) = (Some i)⟩
  using assms
  by simp

```

```

lemma output_layer:
  assumes ⟨y = length i⟩ and ⟨0 < y⟩
  shows ⟨predictlayer_l N (Some i) (Out (⟦name = x, units = y⟧)) = (Some i)⟩
  using assms
  by simp

```

```

lemma dense_layer:
  shows ⟨predictlayer_l N (Some i) (Dense (⟦name = x, units = y, ActivationRecord.φ = p, LayerRecord.β = b, ω = w⟧))
  =
  (if layer_consistentl N (length i) (Dense (⟦name = x, units = y, ActivationRecord.φ = p, LayerRecord.β = b, ω = w⟧)) then
    (let in_w_pairs = map (λ e. zip i e) w;
      wsums = map (λ vs'. ∑ (x,y)←vs'. x*y) in_w_pairs;
      wsum_bias = map (λ (s,b). s+b) (zip wsums b))
    in
    (case activation_tab N p of
      None ⇒ None
      | Some f ⇒ Some (f wsum_bias)))
    else None)⟩
  by simp

```

```

lemma dense_layer':
  assumes ⟨activation_tab N p = Some a⟩
  shows ⟨predictlayer_l N (Some i) (Dense (⟦name = x, units = y, ActivationRecord.φ = p, LayerRecord.β = b, ω = w⟧))
  =
  (if layer_consistentl N (length i) (Dense (⟦name = x, units = y, ActivationRecord.φ = p, LayerRecord.β = b, ω = w⟧)) then
    (let in_w_pairs = map (λ e. zip i e) w;

```

```

      wsums = map (λ vs'. ∑ (x,y)←vs'. x*y) in_w_pairs;
      wsum_bias = map (λ (s,b). s+b) (zip wsums b)
    in Some (a wsum_bias))
  else None)›
using assms
by simp

lemma activation_layer:
assumes ⟨y = length i⟩
shows ⟨predictlayer_l N (Some i) (Activation (|name = x, units = y, ActivationRecord.φ = p|)) =
  (if layer_consistentl N (length i) (Activation (|name = x, units = y, ActivationRecord.φ = p|)) then
    (case activation_tab N p of None ⇒ None | Some f ⇒ Some (f i))
  else None)⟩
using assms
by simp

lemma activation_layer':
assumes ⟨y = length i⟩
and ⟨activation_tab N p = Some a⟩
shows ⟨predictlayer_l N (Some i) (Activation (|name = x, units = y, ActivationRecord.φ = p|)) =
  (if layer_consistentl N (length i) (Activation (|name = x, units = y, ActivationRecord.φ = p|)) then Some (a i) else
  None)⟩
using assms
by simp

lemma predictlayer_l_impl_activation_tab_const: ⟨predictlayer_l_impl N = predictlayer_l_impl (|layers = l, activation_tab = activation_tab N|)⟩
apply(rule ext)+
using neural_network_seq_layers.select_convs predictlayer_l_activation_tab predictlayer_l_impl.elims predictlayer_l_impl.simps
by (smt (verit))

context neural_network_sequential_layersl begin

lemma img_None_1: assumes ⟨(predictseq_layer_l N xs) ≠ None⟩ shows ⟨(length xs = (in_deg_NN N))⟩
proof –
have 1: ⟨(predictseq_layer_l N xs) ≠ None ⟶ (length xs = (in_deg_NN N))⟩
proof(cases N)
case (fields layers' activation_tab') note i = this
then show ?thesis
  unfolding predictseq_layer_l_def
  using i layers_nonempty
proof(induction layers')
case Nil
then show ?case by simp
next
case (Cons a layers')
then show ?case proof(cases a)
  case (In x1)
  then show ?thesis using Cons
  by (simp add: fold_predict_l_strict in_deg_NN_def)
next
case (Out x2)
then show ?thesis using Cons neural_network_sequential_layersl_axioms

```

```

    by (simp add: fold_predict_L_strict in_deg_NN_def)
  next
  case (Dense x3)
  then show ?thesis using Cons neural_network_sequential_layers_l_axioms i
  apply (simp add: fold_predict_L_strict in_deg_NN_def)
  using Cons by (metis isIn.simps(3) list.sel(1) neural_network_seq_layers.select_convs(1)
    neural_network_sequential_layers_l.head_is_In)
  next
  case (Activation x4)
  then show ?thesis using Cons neural_network_sequential_layers_l_axioms i
  by (simp add: fold_predict_L_strict in_deg_NN_def)
qed
qed
qed
show ?thesis using assms 1 by simp
qed

lemma img_None_2':
  assumes a0: ⟨layers' ≠ []⟩
  and a4: ⟨valid_activation_tab_l activation_tab'⟩
  and a1: ⟨layers_consistent_l (layers = [], activation_tab = activation_tab') (length xs) layers'⟩
  shows ⟨foldl (predict_layer_l (layers = [], activation_tab = activation_tab')) (Some xs) layers' ≠ None⟩
proof (insert assms, induction layers' arbitrary: xs rule: list_nonempty_induct)
  case (single l)
  then show ?case
  proof (cases l)
    case (In x1)
    then show ?thesis
    using single by (simp)
  next
  case (Out x2)
  then show ?thesis
  using single by (simp)
  next
  case (Dense x3)
  then show ?thesis
  using single by (force)
  next
  case (Activation x4)
  then show ?thesis
  using single by (force)
qed
next
case (cons l ls)
then show ?case
proof (cases l)
  case (In x1)
  then show ?thesis using cons by simp
next
  case (Out x2)
  then show ?thesis using cons by simp
next
  case (Dense x3)
  then show ?thesis

```

```

using cons assms
apply clarsimp
apply (subst cons.IH[simplified], simp_all)
using valid_activation_preserves_length by force
next
case (Activation x4)
then show ?thesis
  using cons assms apply clarsimp
  apply (subst cons.IH[simplified], simp_all)
  by (metis valid_activation_preserves_length)
qed
qed

```

```

lemma img_None_2:
  assumes <length xs = in_deg_NN N>
  shows <((predict_seq_layer_l N xs) ≠ None)>
proof(cases N)
  case (fields layers activation_tab)
  then show ?thesis
    using assms layer_valid apply simp
    unfolding neural_network_sequential_layers_l_def predict_seq_layer_l_def
    apply (subst predict_layer_l_activation_tab_const[of _ []])
    apply simp
    apply (subst img_None_2'[of layers activation_tab xs, simplified], simp_all)
    using layers_nonempty apply force
    using activation_tab_valid apply fastforce
    using assms layer_valid unfolding in_deg_NN_def apply simp
    using layers_consistent_l_activation_tab_const by fastforce
qed

```

```

lemma img_None: <((predict_seq_layer_l N xs) ≠ None) = (length xs = in_deg_NN N)>
  using img_None_1 img_None_2 by blast

```

```

lemma img_Some: <(∃ y. (predict_seq_layer_l N xs) = Some y) = (length xs = in_deg_NN N)>
  using img_None by simp

```

```

lemma img_length: <(∃ y. ((predict_seq_layer_l N xs) = Some y) → (length y = out_deg_NN N))>
proof(cases N)
  case (fields layers' activation_tab') note i = this
  then show ?thesis
    unfolding predict_seq_layer_l_def i
  proof(induction layers' arbitrary: xs rule:rev_induct)
    case Nil
    then show ?case by auto
  next
    case (snoc a layers')
    then show ?case proof(cases a)
      case (In x1)
      then show ?thesis using snoc
        using Ex_list_of_length by blast
    next
      case (Out x2)
      then show ?thesis using snoc

```

```

    apply (simp add: fold_predict_l_strict out_deg_NN_def neural_network_sequential_layers_l_def)
    using Ex_list_of_length by blast
  next
  case (Dense x3)
  then show ?thesis using Cons neural_network_sequential_layers_l_axioms i
    apply (simp add: fold_predict_l_strict in_deg_NN_def)
    using snoc
    by (smt (verit, ccfv_threshold) Ex_list_of_length)
  next
  case (Activation x4)
  then show ?thesis using Cons neural_network_sequential_layers_l_axioms i
    apply (simp add: fold_predict_l_strict in_deg_NN_def)
    using snoc
    by (smt (verit, ccfv_threshold) Ex_list_of_length)
qed
qed
qed

```

```

lemma predict_layer_l_impl_eq:
  assumes <layer_consistent_l N (length inputs) l>
  shows <predict_layer_l N (Some inputs) l = Some (predict_layer_l_impl N inputs l)>
proof (cases l)
  case (In x1)
  then show ?thesis using assms by (simp)
next
  case (Out x2)
  then show ?thesis using assms by (simp)
next
  case (Dense x3)
  then show ?thesis
    using Dense assms by (force)
next
  case (Activation x4)
  then show ?thesis using Activation assms by (force)
qed

```

```

lemma aux_length: <
  0 < units x3  $\implies$  valid_activation_tab_l atab  $\implies$ 
  inputs  $\neq [] \implies$ 
  length ( $\beta$  x3) = units x3  $\implies$ 
  length ( $\omega$  x3) = units x3  $\implies$ 
   $\forall r \in \text{set } (\omega \text{ x3}). \text{length } r = \text{length inputs} \implies$ 
  atab ( $\varphi$  x3) = Some y  $\implies$ 
  (length (y (map2 (+) (map (( $\lambda$  vs'.  $\sum (x, y) \leftarrow \text{vs}'. x * y$ )  $\circ$  zip inputs) ( $\omega$  x3)) ( $\beta$  x3)))) = units x3
>
  using valid_activation_preserves_length[of atab ( $\varphi$  x3) y, symmetric] by simp

```

```

lemma pred_list_impl_aux':
  assumes <ls  $\neq []$ >
  and layer_valid: <layer_consistent_l (layers = [], activation_tab = atab) (length inputs) ls>
  and activation_tab_valid: <valid_activation_tab_l atab>
  shows <
  foldl (predict_layer_l (layers = [], activation_tab = atab)) (Some inputs) ls =

```

```

    Some (foldl (predictlayer_l_impl (layers = [], activation_tab = atab)) inputs ls)
  >
proof(insert assms(1) assms(2), induction ls arbitrary:inputs rule:list_nonempty_induct)
case (single x)
then show ?case proof(cases x)
  case (In x1)
    then show ?thesis
    using In single by force
  next
    case (Out x2)
    then show ?thesis
    using Out single.prem by force
  next
    case (Dense x3)
    then show ?thesis
    using Dense single by force
  next
    case (Activation x4)
    then show ?thesis
    using Activation single by force
  qed
next
case (cons x xs)
then show ?case proof(cases x)
  case (In x1)
    then show ?thesis
    by (metis (no_types, lifting) cons.IH cons.prem foldl_Cons layer_consistentl.simps(1) layers_consistentl.simps(2)
      layers_consistentl_layersN_const neural_network_seq_layers.select_convs(2) out_deg_layer.simps(1) pre-
      dictlayer_l.simps(1) predictlayer_l_impl.simps(1))
  next
    case (Out x2)
    then show ?thesis
    apply(simp add:cons assms)
    using cons.prem
    cons.IH cons.prem layers_consistentl.simps(2) out_deg_layer.simps(2)
    by simp
  next
    case (Dense x3)
    then show ?thesis
    apply(thin_tac x = Dense x3)
    using cons.prem apply(clarsimp simp add:Dense)
    apply(subst cons.IH, simp_all)
    apply(insert assms(3))
    apply(subst aux_length, simp_all)
    unfolding valid_activation_tabl_def ran_def by auto
  next
    case (Activation x4)
    then show ?thesis
    apply(simp add:cons assms)
    using cons.prem assms(3) cons.IH cons.prem layers_consistentl.simps(2)
      layers_consistentl_layersN_const neural_network_seq_layers.select_convs(2) out_deg_layer.simps(3)
      valid_activation_preserves_length
    by (smt (verit, del_insts) layer_consistentl.simps(3) neqo_conv option.sel option.simps(5))
  qed

```

qed

lemma pred_list_impl_aux:

```
assumes layer_valid: <layers_consistent_l (layers = ls, activation_tab = atab) (length inputs) ls>
  and activation_tab_valid: <valid_activation_tab_l atab>
shows <
  foldl (predict_layer_l (layers = ls, activation_tab = atab)) (Some inputs) ls =
  Some (foldl (predict_layer_l_impl (layers = ls, activation_tab = atab)) inputs ls)
  >
proof(cases ls)
case Nil
then show ?thesis by simp
next
case (Cons a list)
then show ?thesis
  using pred_list_impl_aux' assms layers_consistent_l_activation_tab_const list.discr
  neural_network_seq_layers.select_convs(2) predict_layer_l_activation_tab_const
  by (metis predict_layer_l_activation_tab_const predict_layer_l_impl_activation_tab_const)
qed
```

lemma predict_seq_layer_l_code [code]:

```
assumes <in_deg_NN N = length inputs>
shows <predict_seq_layer_l N inputs = Some (predict_seq_layer_l_impl N inputs)>
proof(cases N)
case (fields ls atab)
then show ?thesis
  unfolding predict_seq_layer_l_def predict_seq_layer_l_impl_def
  using assms apply (simp, subst pred_list_impl_aux)
  using layer_valid apply force
  using activation_tab_valid apply force
  by simp
qed
```

lemma predict_seq_layer_l_code' [code]:

```
assumes <in_deg_NN N = length inputs>
shows <the (predict_seq_layer_l N inputs) = predict_seq_layer_l_impl N inputs>
by (simp add: assms predict_seq_layer_l_code)
```

end

ML<

```
val layer_config_list = {
  InC = @ {const NN_Layers.layer.In (<real list>, <activation_multi>, <real list list>)},
  OutC = @ {const NN_Layers.layer.Out (<real list>, <activation_multi>, <real list list>)},
  DenseC = @ {const NN_Layers.layer.Dense (<activation_multi>, <real list>, <real list list>)},
  InOutRecordC = @ {const NN_Layers.InOutRecord.InOutRecord_ext (<activation_multi>, (real list, real list list,
unit) LayerRecord_ext) ActivationRecord_ext)},
  LayerRecordC = @ {const LayerRecord_ext (<real list>, <real list list>, <unit>)},
  ActivationRecordC = @ {const ActivationRecord_ext (<activation_multi>, (<real list>, real list list, unit) LayerRecord_ext)},
  biasT_conv = (fn x => x),
  weightsT_conv = (fn x => fn _ => x),
```

```

ltype = @{typ <(real list, activationmulti, real list list) layer>},
activation_term = Activation_Term.MultiList,
layersT = @{typ <((real list, activationmulti, real list list) layer) list>},
phiT = @{typ <activationmulti ⇒ (real list ⇒ real list) option>},
layer_extC = @{const <NN_Layers.neural_network_seq_layers.neural_network_seq_layers_ext> (⟨real list⟩, ⟨ac-
tivationmulti⟩, ⟨real list list⟩, ⟨unit⟩)},
layer_def = @{thm neural_network_sequential_layersl_def},
valid_activation_tab = @{thm valid_activation_tabl_def},
locale_name = neural_network_sequential_layersl
}:Convert_TensorFlow_Seq_Layer.layer_config
val _ = Theory.setup
  (Convert_TensorFlow_Symtab.add_encoding(seq_layer_list,
    Convert_TensorFlow_Seq_Layer.def_seq_layer_nn layer_config_list))
>
end

```

5.2.3 Neural Network as Sequential Layers using Vector Spaces (≡ NN_Layers_Matrix_Main)

theory

NN_Layers_Matrix_Main

imports

NN_Lipschitz_Continuous

NN_Layers

Matrix_Utils

Properties_Matrix

begin

In this theory, we model feed-forward neural networks as “computational layers” following the structure of TensorFlow [1] closely.

definition $\langle \text{valid_activation_tab}_m \text{ tab} = (\forall f \in \text{ran } \text{tab}. \forall \text{xs}. \text{dim_vec } \text{xs} = \text{dim_vec } (f \text{ xs})) \rangle$

lemma *valid_activation_preserves_dim*:

assumes $\langle \text{valid_activation_tab}_m \text{ t} \rangle$

assumes $\langle \text{t } n = \text{Some } f \rangle$

shows $\langle \text{dim_vec } \text{xs} = \text{dim_vec } (f \text{ xs}) \rangle$

using *assms unfolding valid_activation_tab_m_def*

using *ranI by metis*

fun *layer_consistent_m* :: $(\text{'a } \text{vec}, \text{'b}, \text{'c } \text{mat}) \text{ neural_network_seq_layers} \Rightarrow \text{nat} \Rightarrow (\text{'a } \text{vec}, \text{'b}, \text{'c } \text{mat}) \text{ layer} \Rightarrow \text{bool}$

where

$\langle \text{layer_consistent}_m \text{ } n \text{c } (\text{In } l) = (o < \text{units } l \wedge n \text{c} = \text{units } l) \rangle$

$\langle \text{layer_consistent}_m \text{ } n \text{c } (\text{Out } l) = (o < \text{units } l \wedge n \text{c} = \text{units } l) \rangle$

$\langle \text{layer_consistent}_m \text{ } N \text{ } n \text{c } (\text{Activation } l) = ((o < \text{units } l \wedge n \text{c} = \text{units } l) \wedge ((\text{activation_tab } N) (\varphi l) \neq \text{None})) \rangle$

$\langle \text{layer_consistent}_m \text{ } N \text{ } n \text{c } (\text{Dense } l) = (o < \text{units } l \wedge o < n \text{c} \wedge \text{dim_vec } (\beta l) = \text{units } l \wedge \text{dim_col } (\omega l) = \text{units } l \wedge \text{dim_row } (\omega l) = n \text{c} \wedge ((\text{activation_tab } N) (\varphi l) \neq \text{None})) \rangle$

fun *layers_consistent_m* **where**

$\langle \text{layers_consistent}_m \text{ } N \text{ } [] = \text{True} \rangle$

$\langle \text{layers_consistent}_m \text{ } N \text{ } w \text{ } (l \# l \text{s}) = ((\text{layer_consistent}_m \text{ } N \text{ } w \text{ } l) \wedge (\text{layers_consistent}_m \text{ } N \text{ } (\text{out_deg_layer } l) \text{ } l \text{s})) \rangle$

```

lemma layer_consistentm_activation_tab_const:
  ⟨layer_consistentm N nc l = layer_consistentm (|layers = ls, activation_tab = activation_tab N|) nc l⟩
  by(cases l, simp_all)

lemma layers_consistentm_activation_tab_const:
  ⟨layers_consistentm N nc ls = layers_consistentm (|layers = ls', activation_tab = activation_tab N|) nc ls⟩
  proof(induction ls arbitrary: nc)
  case Nil
  then show ?case by simp
  next
  case (Cons a ls)
  then show ?case
  by(simp add: layer_consistentm_activation_tab_const[symmetric])
qed

lemma layers_consistentmAll:
  assumes ⟨layers_consistentm N inputs (layers N)⟩
  shows ⟨∀ l ∈ set (layers N). ∃ n . layer_consistentm N n l⟩
  proof(cases N)
  case (fields layers activation_tab) note i = this
  then show ?thesis
  apply(insert assms, simp add: i, safe)
  apply(thin_tac N = (|layers = layers, activation_tab = activation_tab|))
  apply(subst layer_consistentm_activation_tab_const[of _ _ []])
  apply(induction layers arbitrary:inputs activation_tab)
  apply(simp)
  using layers_consistentm_activation_tab_const layer_consistentm_activation_tab_const[of _ _ []]
  apply(clarsimp)[1]
  using layer_consistentm_activation_tab_const by fastforce
qed

lemma layers_consistentmAll':
  assumes ⟨layers_consistentm N (in_deg_NN N) (layers N)⟩
  shows ⟨∀ l ∈ set (layers N). ∃ n . layer_consistentm N n l⟩
  using assms layers_consistentmAll by blast

locale neural_network_sequential_layersm =
  fixes N::⟨'a::comm_ring Matrix.vec, 'b, 'a Matrix.mat⟩ neural_network_seq_layers⟩
  assumes head_is_In: ⟨isIn (hd (layers N))⟩
  and last_is_Out: ⟨isOut (last (layers N))⟩
  and layer_internal: ⟨list_all isInternal ((tl o butlast) (layers N))⟩
  and activation_tab_valid: ⟨valid_activation_tabm (activation_tab N)⟩
  and layer_valid: ⟨layers_consistentm N (in_deg_NN N) (layers N)⟩
begin

lemma layers_nonempty: ⟨layers N ≠ []⟩
  by (metis hd_Nil_eq_last head_is_In isIn.elims(2) isOut.simps(2) last_is_Out)

lemma min_length_layers_two: ⟨1 < length (layers N)⟩
  by (metis (no_types, lifting) One_nat_def add_o append_Nil append_butlast_last_id
  append_eq_append_conv head_is_In isIn.simps(2) isOut.elims(2) last_is_Out
  layers_nonempty length_o_conv less_one linorder_neqE_nat list.sel(1) list.size(4))

```

```

lemma layers_structure:  $\langle \exists \text{ il ol ls. layers } N = (\text{In } \text{il}) \# \text{ls} @ [\text{Out } \text{ol}] \rangle$ 
by (metis (no_types, lifting) append_butlast_last_id head_is_In isln.elims(2) isOut.elims(2)
last.simps last_is_Out layer.distinct(1) layers_nonempty list.exhaust list.sel(1))
end

```

```

fun predict_layer_m ::  $\langle ('a :: \text{comm\_ring } \text{Matrix.vec}, 'b, 'a \text{ Matrix.mat}) \text{neural\_network\_seq\_layers} \Rightarrow ('a \text{ Matrix.vec}) \text{option}$ 
 $\Rightarrow ('a \text{ Matrix.vec}, 'b, 'a \text{ Matrix.mat}) \text{layer} \Rightarrow ('a \text{ Matrix.vec}) \text{option} \rangle$  where
   $\langle \text{predict\_layer\_m } N (\text{Some } \text{vs}) (\text{In } \text{l}) = (\text{if } \text{layer\_consistent}_m \text{ } N (\text{dim\_vec } \text{vs}) (\text{In } \text{l}) \text{ then } \text{Some } \text{vs} \text{ else } \text{None}) \rangle$ 
   $\mid \langle \text{predict\_layer\_m } N (\text{Some } \text{vs}) (\text{Out } \text{l}) = (\text{if } \text{layer\_consistent}_m \text{ } N (\text{dim\_vec } \text{vs}) (\text{Out } \text{l}) \text{ then } \text{Some } \text{vs} \text{ else } \text{None}) \rangle$ 
   $\mid \langle \text{predict\_layer\_m } N (\text{Some } \text{vs}) (\text{Dense } \text{pl}) = (\text{if } \text{layer\_consistent}_m \text{ } N (\text{dim\_vec } \text{vs}) (\text{Dense } \text{pl}) \text{ then}$ 
    (case activation_tab N ( $\varphi$  pl) of
      None  $\Rightarrow$  None
       $\mid$  Some f  $\Rightarrow$  Some (f ((vs v*  $\omega$  pl) +  $\beta$  pl) )
    ) else None  $\rangle$ 
   $\mid \langle \text{predict\_layer\_m } N (\text{Some } \text{vs}) (\text{Activation } \text{pl}) = (\text{if } \text{layer\_consistent}_m \text{ } N (\text{dim\_vec } \text{vs}) (\text{Activation } \text{pl}) \text{ then}$ 
    (case activation_tab N ( $\varphi$  pl) of
      None  $\Rightarrow$  None
       $\mid$  Some f  $\Rightarrow$  Some (f vs)
    ) else None  $\rangle$ 
   $\mid \langle \text{predict\_layer\_m } \_ \text{None } \_ = \text{None} \rangle$ 

```

```

fun
  predict_layer_m_impl ::  $\langle ('a :: \{\text{comm\_ring}\} \text{Matrix.vec}, 'b, 'a \text{ Matrix.mat}) \text{neural\_network\_seq\_layers} \Rightarrow 'a \text{ Matrix.vec}$ 
 $\Rightarrow ('a \text{ Matrix.vec}, 'b, 'a \text{ Matrix.mat}) \text{layer} \Rightarrow 'a \text{ Matrix.vec} \rangle$ 
where
   $\langle \text{predict\_layer\_m\_impl } N \text{ vs } (\text{In } \text{l}) = \text{vs} \rangle$ 
   $\mid \langle \text{predict\_layer\_m\_impl } N \text{ vs } (\text{Out } \text{l}) = \text{vs} \rangle$ 
   $\mid \langle \text{predict\_layer\_m\_impl } N \text{ vs } (\text{Dense } \text{pl}) = ((\text{the } (\text{activation\_tab } N (\varphi \text{ pl}))) ((\text{vs } \text{v} * \omega \text{ pl}) + \beta \text{ pl})) \rangle$ 
   $\mid \langle \text{predict\_layer\_m\_impl } N \text{ vs } (\text{Activation } \text{pl}) = (\text{the } (\text{activation\_tab } N (\varphi \text{ pl})) \text{ vs}) \rangle$ 

```

```

lemma predict_layer_Some:
  assumes  $\langle (\text{layer\_consistent}_m \text{ } N (\text{dim\_vec } \text{xs}) \text{ l}) \rangle$ 
  shows  $\langle (\text{predict\_layer\_m } N (\text{Some } \text{xs}) \text{ l} \neq \text{None}) \rangle$ 
proof(cases l)
  case (In x1)
  then show ?thesis using assms by (simp)
next
  case (Out x2)
  then show ?thesis using assms by simp
next
  case (Dense x3)
  then show ?thesis using assms by force
next
  case (Activation x4)
  then show ?thesis using assms by force
qed

```

```

definition  $\langle \text{predict\_seq\_layer\_m } N \text{ inputs} = \text{foldl } (\text{predict\_layer\_m } N) (\text{Some } \text{inputs}) (\text{layers } N) \rangle$ 

```

```

definition  $\langle \text{predict\_seq\_layer\_m\_impl } N \text{ inputs} = \text{foldl } (\text{predict\_layer\_m\_impl } N) \text{ inputs } (\text{layers } N) \rangle$ 

```

```

definition  $\langle \text{predict\_seq\_layer\_m } 'N \text{ inputs} = \text{map\_option } \text{list\_of\_vec } (\text{predict\_seq\_layer\_m } N (\text{vec\_of\_list } \text{inputs})) \rangle$ 

```

```

lemma predictlayer_l_impl_activation_tab_const: ⟨predictlayer_m_impl N = predictlayer_m_impl (|layers = l, activation_tab = activation_tab N)|⟩
  apply(rule ext)+
  using neural_network_seq_layers.select_convs predictlayer_m_impl.elims predictlayer_m_impl.simps
  by (smt (verit))

```

```

lemma layers_consistentm_layersN_const:
  ⟨layers_consistentm N = layers_consistentm (|layers = ls', activation_tab = activation_tab N)|⟩
  using layers_consistentm_activation_tab_const by blast

```

```

lemma predictlayer_m_impl_eq:
  assumes ⟨layer_consistentm N (dim_vec inputs) l⟩
  shows ⟨predictlayer_m N (Some inputs) l = Some (predictlayer_m_impl N inputs l)⟩
proof(cases l)
  case (In x1)
  then show ?thesis using assms by(simp)
next
  case (Out x2)
  then show ?thesis using assms by(simp)
next
  case (Dense x3)
  then show ?thesis
    using Dense assms by(force)
next
  case (Activation x4)
  then show ?thesis using Activation assms by(force)
qed

```

```

lemma valid_activation_preserves_length:
  assumes ⟨valid_activation_tabm t⟩
  assumes ⟨t n = Some f⟩
  shows ⟨dim_vec xs = dim_vec (f xs)⟩
  using assms unfolding valid_activation_tabm_def
  by (simp add: ranI)

```

```

lemma fold_predict_m_strict: ⟨(foldl (predictlayer_m N) None ls) = None⟩
  by(induction ls, simp_all)

```

```

lemmas [nn_layer] = predictlayer_m.simps predict_layer_Some fold_predict_m_strict

```

```

lemma predictlayer_m_activation_tab: assumes activation_tab N = activation_tab N' shows
  ⟨predictlayer_m N x xs = predictlayer_m N' x xs⟩
proof(cases xs)
  case (In x1)
  then show ?thesis
    by (metis layer_consistentm.simps(1) option.collapse predictlayer_m.simps(1) predictlayer_m.simps(5))
next
  case (Out x2)
  then show ?thesis
    by (metis layer_consistentm.simps(2) option.exhaust predictlayer_m.simps(2) predictlayer_m.simps(5))

```

```

next
case (Dense x3)
then show ?thesis proof(cases x)
  case None
  then show ?thesis by simp
next
case (Some a)
then show ?thesis using Dense assms by force
qed
next
case (Activation x4)
then show ?thesis proof(cases x)
  case None
  then show ?thesis by simp
next
case (Some a)
then show ?thesis using Activation assms by force
qed
qed

lemma predictlayer_m_activation_tab_const: ⟨predictlayer_m N = predictlayer_m (|layers = l, activation_tab = activation_tab N)⟩
  apply(rule ext)+
  by (metis neural_network_seq_layers.select_convs(2) predictlayer_m_activation_tab)

context neural_network_sequential_layersm begin
lemma img_None_1: assumes ⟨(predictseq_layer_m N xs) ≠ None⟩ shows ⟨(dim_vec xs = (in_deg_NN N))⟩
proof –
have ⟨(predictseq_layer_m N xs) ≠ None ⟶ (dim_vec xs = (in_deg_NN N))⟩
proof(cases N)
case (fields layers' activation_tab') note i = this
then show ?thesis
  unfolding predictseq_layer_m_def
  using i layers_nonempty
  proof(induction layers')
  case Nil
  then show ?case by simp
next
case (Cons a layers')
then show ?case proof(cases a)
  case (In x1)
  then show ?thesis using Cons
  by (simp add: fold_predict_m_strict in_deg_NN_def)
next
case (Out x2)
then show ?thesis using Cons
  by (simp add: fold_predict_m_strict in_deg_NN_def)
next
case (Dense x3)
then show ?thesis using Cons neural_network_sequential_layersm_axioms i
  apply(simp add: fold_predict_m_strict in_deg_NN_def)
  using Cons by (metis isIn.simps(3) list.sel(1) neural_network_seq_layers.select_convs(1)
    neural_network_sequential_layersm.head_is_In)
next

```

```

    case (Activation x4)
    then show ?thesis using Cons neural_network_sequential_layers_m_axioms i
    by(simp add: fold_predict_m_strict in_deg_NN_def)
  qed
qed
qed
then show ?thesis using assms by simp
qed

lemma img_None_2':
  assumes a0: ⟨layers' ≠ []⟩
    and a4: ⟨valid_activation_tab_m activation_tab'⟩
    and a1: ⟨layers_consistent_m (layers = [], activation_tab = activation_tab') (dim_vec xs) layers'⟩
  shows ⟨foldl (predict_layer_m (layers = [], activation_tab = activation_tab')) (Some xs) layers' ≠ None⟩
proof(insert assms, induction layers' arbitrary: xs rule:list_nonempty_induct)
  case (single l)
  then show ?case
  proof(cases l)
    case (In x1)
    then show ?thesis
    using single by(simp)
  next
    case (Out x2)
    then show ?thesis
    using single by(force)
  next
    case (Dense x3)
    then show ?thesis
    using single by(force)
  next
    case (Activation x4)
    then show ?thesis
    using single by(force)
  qed
next
  case (cons l ls)
  then show ?case
  proof(cases l)
    case (In x1)
    then show ?thesis using cons by force
  next
    case (Out x2)
    then show ?thesis using cons by force
  next
    case (Dense x3)
    then show ?thesis
    using cons assms apply(clarsimp)[1]
    apply(subst cons.IH[simplified], simp_all)
    using valid_activation_preserves_dim by force
  next
    case (Activation x4)
    then show ?thesis
    using cons assms apply(clarsimp)[1]
    apply(subst cons.IH[simplified], simp_all)

```

```

  by (metis valid_activation_preserves_dim)
qed
qed

```

```

lemma img_None_2:
  assumes <dim_vec xs = in_deg_NN N>
  shows <((predict_seq_layer_m N xs) ≠ None)>
proof(cases N)
  case (fields layers activation_tab)
  then show ?thesis
  using assms layer_valid apply(simp)
  unfolding neural_network_sequential_layers_m_def predict_seq_layer_m_def
  apply(subst predict_layer_m_activation_tab_const[of _ []])
  apply(simp)
  apply(subst img_None_2'[of layers activation_tab xs, simplified], simp_all)
  using layers_nonempty apply force
  using activation_tab_valid apply fastforce
  using assms layer_valid unfolding in_deg_NN_def apply(simp)
  using layers_consistent_m_activation_tab_const by fastforce
qed

```

```

lemma img_None: <((predict_seq_layer_m N xs) ≠ None) = (dim_vec xs = in_deg_NN N)>
  using img_None_1 img_None_2 by blast

```

```

lemma img_Some: <(∃ y. (predict_seq_layer_m N xs) = Some y) = (dim_vec xs = in_deg_NN N)>
  using img_None by simp

```

```

lemma img_deg: <(∃ y. ((predict_seq_layer_m N xs) = Some y) → (dim_vec y = out_deg_NN N))>
proof(cases N)
  case (fields layers' activation_tab')
  then show ?thesis
  unfolding fields_predict_seq_layer_m_def
  using fields_layers_nonempty
proof(induction layers' arbitrary: xs rule:rev_induct)
  case Nil
  then show ?case by simp
next
  case (snoc a layers')
  then show ?case proof(cases a)
    case (In x1)
    then show ?thesis using snoc
    using dim_vec by blast
  next
    case (Out x2)
    then show ?thesis using snoc
    apply (simp add: fold_predict_m_strict_out_deg_NN_def neural_network_sequential_layers_m_def)
    using dim_vec_first by blast
  next
    case (Dense x3)
    then show ?thesis using Cons_neural_network_sequential_layers_m_axioms
    apply (simp add: fold_predict_m_strict_in_deg_NN_def)
    using snoc
    by (simp add: neural_network_sequential_layers_m_def)
  next

```

```

case (Activation x4)
then show ?thesis using Cons neural_network_sequential_layers_m_axioms
  apply(simp add: fold_predict_m_strict in_deg_NN_def)
  using snoc
  using snoc
  by (simp add: neural_network_sequential_layers_m_def)
qed
qed
qed

lemma aux_length: 0 < units x3  $\implies$ 
  0 < dim_vec inputs  $\implies$ 
  dim_vec ( $\beta$  x3) = units x3  $\implies$ 
  dim_col ( $\omega$  x3) = units x3  $\implies$ 
  dim_row ( $\omega$  x3) = dim_vec inputs  $\implies$ 
  layers_consistent_m ( $\lambda$ layers = [], activation_tab = atab) (units x3) xs  $\implies$ 
  atab ( $\varphi$  x3) = Some y  $\implies$  valid_activation_tab_m atab  $\implies$  layers_consistent_m ( $\lambda$ layers = [], activation_tab = atab)
  (dim_vec (y (inputs v *  $\omega$  x3 +  $\beta$  x3))) xs
  using valid_activation_preserves_length[of atab ( $\varphi$  x3) y, symmetric] by simp

lemma pred_mat_impl_aux':
  assumes <ls  $\neq$  []>
  and layer_valid: <layers_consistent_m ( $\lambda$ layers = [], activation_tab = atab) (dim_vec inputs) ls>
  and activation_tab_valid: <valid_activation_tab_m atab>
  shows <
    foldl (predict_layer_m ( $\lambda$ layers = [], activation_tab = atab)) (Some inputs) ls =
    Some (foldl (predict_layer_m_impl ( $\lambda$ layers = [], activation_tab = atab)) inputs ls)
  >

proof(insert assms(1) assms(2), induction ls arbitrary:inputs rule:list_nonempty_induct)
case (single x)
then show ?case proof(cases x)
  case (In x1)
  then show ?thesis
  using In single by force
next
  case (Out x2)
  then show ?thesis
  using Out single.premis by force
next
  case (Dense x3)
  then show ?thesis
  using Dense single by force
next
  case (Activation x4)
  then show ?thesis
  using Activation single by force
qed
next
case (cons x xs)
then show ?case proof(cases x)
  case (In x1)
  then show ?thesis
  by (metis (no_types, lifting) cons.IH cons.premis foldl_Cons layer_consistent_m_simps(1) layers_consistent_m_simps(2))

```

```

      layers_consistentm.layersN_const neural_network_seq_layers.select_convs(2) out_deg_layer.simps(1) predictlayer-m.simps(1) predictlayer-m-impl.simps(1))
next
  case (Out x2)
  then show ?thesis
  apply(clarsimp simp add:cons assms)[1]
  using cons.IH cons.premis layers_consistentm.simps(2) out_deg_layer.simps(2)
  cons.premis cons(2,3)
  by fastforce
next
  case (Dense x3)
  then show ?thesis
  apply(thin_tac x = Dense x3)
  using cons.premis apply(clarsimp simp add:Dense)[1]
  apply(subst cons.IH, simp_all)
  apply(insert assms(3))
  by(subst aux_length, simp_all)
next
  case (Activation x4)
  then show ?thesis
  apply(clarsimp simp add:cons assms)[1]
  using assms(3) cons.IH cons.premis layers_consistentm.simps(2)
  layers_consistentm.layersN_const neural_network_seq_layers.select_convs(2) out_deg_layer.simps(3)
  valid_activation_preserves_length
  by (smt (z3) layer_consistentm.simps(3) neqo_conv option.case_eq_if option.sel)

qed
qed

```

```

lemma pred_mat_impl_aux:
  assumes layer_valid: <layers_consistentm (layers = ls, activation_tab = atab) (dim_vec inputs) ls>
  and activation_tab_valid: <valid_activation_tabm atab>
  shows <
    foldl (predictlayer-m (layers = ls, activation_tab = atab)) (Some inputs) ls =
    Some (foldl (predictlayer-m-impl (layers = ls, activation_tab = atab)) inputs ls)
  >
proof(cases ls)
  case Nil
  then show ?thesis by simp
next
  case (Cons a list)
  then show ?thesis
  using pred_mat_impl_aux' assms layers_consistentm.activation_tab_const list.discr
  neural_network_seq_layers.select_convs(2) predictlayer-m.activation_tab_const
  by (metis predictlayer-m.activation_tab_const predictlayer-l.impl_activation_tab_const)
qed

```

```

lemma predictseq_layer-m_code [code]:
  assumes <in_deg_NN N = dim_vec inputs>
  shows <predictseq_layer-m N inputs = Some (predictseq_layer-m-impl N inputs)>
proof(cases N)
  case (fields ls atab)
  then show ?thesis

```

```

unfolding predict_seq_layer_m_def predict_seq_layer_m_impl_def
using assms apply(simp,subst pred_mat_impl_aux)
using layer_valid apply force
using activation_tab_valid apply force
by simp
qed

lemma predict_seq_layer_m_code' [code]:
  assumes ‹in_deg_NN N = dim_vec inputs›
  shows ‹the (predict_seq_layer_m N inputs) = predict_seq_layer_m_impl N inputs›
  by (simp add: assms predict_seq_layer_m_code)

end

```

ML

```

val layer_config_matrix = {
  InC = @ {const NN_Layers.layer.In(⟨real Matrix.vec⟩, ⟨activation_multi⟩, ⟨real Matrix.mat⟩)},
  OutC = @ {const NN_Layers.layer.Out(⟨real Matrix.vec⟩, ⟨activation_multi⟩, ⟨real Matrix.mat⟩)},
  DenseC = @ {const NN_Layers.layer.Dense(⟨activation_multi⟩,⟨real Matrix.vec⟩, ⟨real Matrix.mat⟩)},
  InOutRecordC = @ {const NN_Layers.InOutRecord.InOutRecord_ext(⟨(activation_multi, (real Matrix.vec, real Matrix.mat, unit)LayerRecord_ext) ActivationRecord_ext⟩)},
  LayerRecordC = @ {const LayerRecord_ext(⟨real Matrix.vec⟩, ⟨real Matrix.mat⟩, ⟨unit⟩)},
  ActivationRecordC = @ {const ActivationRecord_ext (⟨activation_multi⟩,⟨(real Matrix.vec, real Matrix.mat, unit)LayerRecord_ext⟩)},
  biasT_conv = (fn x => @ {const vec_of_list(⟨real⟩)} $ x),
  weightsT_conv = (fn x => fn dim => @ {const mat_of_cols_list(⟨real⟩)} $ HLogic.mk_number typ ‹nat› dim $ x),
  ltype = @ {typ ‹(real Matrix.vec, activation_multi, real Matrix.mat) layer⟩},
  activation_term = Activation_Term.MultiMatrix,
  layersT = @ {typ ‹((real Matrix.vec, activation_multi, real Matrix.mat) layer) list⟩},
  phiT = @ {typ ‹activation_multi ⇒ (real Matrix.vec ⇒ real Matrix.vec) option⟩},
  layer_extC = @ {const ‹NN_Layers.neural_network_seq_layers.neural_network_seq_layers_ext⟩ (⟨real Matrix.vec⟩,⟨activation_multi⟩, ⟨real Matrix.mat⟩, ⟨unit⟩)},
  layer_def = @ {thm neural_network_sequential_layers_m_def},
  valid_activation_tab = @ {thm valid_activation_tab_m_def},
  locale_name = neural_network_sequential_layers_m
}:Convert_TensorFlow_Seq_Layer.layer_config
val _ = Theory.setup
  (Convert_TensorFlow_Symtab.add_encoding(seq_layer_matrix,
    Convert_TensorFlow_Seq_Layer.def_seq_layer_nn layer_config_matrix))
}
end

```

5.3 Main Theory (Layers) (NN_Layers_Main)

```

theory
  NN_Layers_Main
  imports
    NN_Common
    Activation_Functions
    NN_Digraph_Layers
    NN_Layers_List_Main

```

NN_Layers_Matrix_Main

begin

5.3.1 Converting between List-based and Matrix-based Sequential Layer Models

```
fun layer_list_to_matrix::⟨('a list, 'b, 'a list list) layer ⇒ ('a Matrix.vec, 'b, 'a Matrix.mat) layer⟩
  where
    ⟨layer_list_to_matrix (In l) = (In l)⟩
  | ⟨layer_list_to_matrix (Out l) = (Out l)⟩
  | ⟨layer_list_to_matrix (Activation l) = (Activation (|name = name l, units = units l, φ = φ l|))⟩
  | ⟨layer_list_to_matrix (Dense l) = (let dimc = length (List.hd (ω l)) in
    (Dense (|name = name l, units = units l, φ = φ l,
    β = vec_of_list (β l), ω = transpose_mat (mat_of_rows_list dimc (ω l)) |))⟩
```

fun

```
layer_matrix_to_list::⟨('a Matrix.vec, 'b, 'a Matrix.mat) layer ⇒ ('a list, 'b, 'a list list) layer⟩
  where
    ⟨layer_matrix_to_list (In l) = (In l)⟩
  | ⟨layer_matrix_to_list (Out l) = (Out l)⟩
  | ⟨layer_matrix_to_list (Activation l) = (Activation (|name = name l, units = units l, φ = φ l|))⟩
  | ⟨layer_matrix_to_list (Dense l) = (Dense (|name = name l, units = units l, φ = φ l,
    β = list_of_vec (β l), ω = mat_to_list (transpose_mat (ω l)) |))⟩
```

definition activation_list_to_matrix::⟨('b ⇒ ((('a list ⇒ 'a list) option)) ⇒ ('b ⇒ ((('a Matrix.vec ⇒ 'a Matrix.vec) option)))⟩

where

```
activation_list_to_matrix a = map_option (λ f . vec_of_list ∘ f ∘ list_of_vec) ∘ a
```

definition activation_matrix_to_list::⟨('b ⇒ ((('a Matrix.vec ⇒ 'a Matrix.vec) option)) ⇒ ('b ⇒ ((('a list ⇒ 'a list) option)))⟩

where

```
activation_matrix_to_list a = map_option (λ f . list_of_vec ∘ f ∘ vec_of_list) ∘ a
```

definition

nn_list_to_matrix::⟨('a list, 'b, 'a list list) neural_network_seq_layers ⇒ ('a Matrix.vec, 'b, 'a mat) neural_network_seq_layers

where

```
⟨nn_list_to_matrix N = (|layers = map layer_list_to_matrix (layers N),
  activation_tab = activation_list_to_matrix (activation_tab N)|)⟩
```

definition

nn_matrix_to_list::⟨('a Matrix.vec, 'b, 'a mat) neural_network_seq_layers ⇒ ('a list, 'b, 'a list list) neural_network_seq_layers

where

```
⟨nn_matrix_to_list N = (|layers = map layer_matrix_to_list (layers N),
  activation_tab = activation_matrix_to_list (activation_tab N)|)⟩
```

5.3.2 Converting Between List/Matrix-based Representations Preserves Consistency

lemma layer_list_matrix_inverse:

```
⟨layer_consistentl N n l ⇒ layer_matrix_to_list (layer_list_to_matrix l) = l⟩
```

```

proof(induction l)
  case (In x)
  then show ?case by simp
next
  case (Out x)
  then show ?case by simp
next
  case (Dense x) note i = this
  then show ?case
    apply(simp add:list_vec)
    apply(subst mat_list[of ω x])
    apply (metis empty_iff list.set(1) list.set_sel(1))
    by(simp)
next
  case (Activation x)
  then show ?case by simp
qed

```

lemma *layer_list_list_inverse*:

$\langle \text{layer_consistent}_m \ N \ n \ l \implies \text{layer_list_to_matrix} (\text{layer_matrix_to_list } l) = l \rangle$

```

proof(induction l)
  case (In x)
  then show ?case by simp
next
  case (Out x)
  then show ?case by simp
next
  case (Dense x) note ii = this
  then show ?case
proof(cases  $\forall c \in \text{set} (\text{mat\_to\_list } (\omega x)^T). \text{dim\_row } (\omega x) = (\text{length } c)$ )
  case True note i = this
  then show ?thesis proof(cases  $\text{mat\_to\_list } (\omega x)^T = []$ )
    case True
    then show ?thesis using i ii apply(simp add:vec_list Let_def)
      by (metis dim_row_list index_transpose_mat(2) less_nat_zero_code list.size(3))
    next
    case False
    then have  $\langle \text{dim\_row } (\omega x) = \text{length} (\text{List.hd } (\text{mat\_to\_list } (\omega x)^T)) \rangle$  using i by simp
    then show ?thesis using i
      using i list_mat_transpose_transpose list_mat index_transpose_mat Matrix.transpose_transpose
      list_mat_transpose_transpose[of ω x, simplified]
      by(simp add:vec_list)
    qed
  next
  case False
  then show ?thesis
    by (metis dim_col_mat_list index_transpose_mat(3))
  qed
next
  case (Activation x)
  then show ?case by simp
qed

```

```

lemma activation_list_inverse: <activation_matrix_to_list (activation_list_to_matrix a) x = a x>
proof(cases a x = None)
  case True
  then show ?thesis
    unfolding activation_list_to_matrix_def activation_matrix_to_list_def o_def
    by simp
  next
  case False
  then show ?thesis
    unfolding activation_list_to_matrix_def activation_matrix_to_list_def o_def
    by(auto simp add:list_vec)
qed

```

```

lemma activation_list_inverse': <activation_matrix_to_list (activation_list_to_matrix a) = a>
by(rule ext, metis activation_list_inverse)

```

```

lemma activation_matrix_inverse: <activation_list_to_matrix (activation_matrix_to_list a) x = a x>
proof(cases a x = None)
  case True
  then show ?thesis
    unfolding activation_list_to_matrix_def activation_matrix_to_list_def o_def
    by simp
  next
  case False
  then show ?thesis
    unfolding activation_list_to_matrix_def activation_matrix_to_list_def o_def
    by(auto simp add:vec_list)
qed

```

```

lemma activation_matrix_inverse': <activation_list_to_matrix (activation_matrix_to_list a) = a>
by(rule ext, metis activation_matrix_inverse)

```

```

lemma is_In_seq_L_eq_m:
  assumes <(layers N) ≠ []>
  shows <isIn (List.hd (layers N)) = isIn (List.hd (layers (nn_list_to_matrix N)))>
proof(cases N)
  case (fields layers activation_tab) note i = this
  then show ?thesis
  proof(cases layers)
    case Nil
    then show ?thesis using assms i by simp
  next
  case (Cons a list)
  then show ?thesis
    using assms i
    unfolding nn_list_to_matrix_def
    by(cases a, simp_all)
  qed
qed

```

```

lemma is_Out_seq_L_eq_m:
  assumes <(layers N) ≠ []>

```

```

shows <isOut (last (layers N)) = isOut (last (layers (nn_list_to_matrix N)))>
proof(cases N)
case (fields layers activation_tab) note i = this
then show ?thesis
  using assms unfolding fields
proof(induction layers)
  case Nil
  then show ?case by simp
next
  case (Cons a layers)
  then show ?case
unfolding nn_list_to_matrix_def
  by(cases a, simp_all)
qed
qed

```

```

lemma is_Internal_seq_L_eq_m:
  assumes <(layers N) ≠ []>
  shows <list_all isInternal ((List.tl o butlast) (layers N)) = list_all isInternal ((List.tl o butlast) (layers (nn_list_to_matrix N)))>
proof(cases N)
case (fields layers activation_tab) note i = this
then show ?thesis
  using assms unfolding fields
proof(induction layers)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)

  then show ?case proof(induction xs)
    case Nil
    then show ?case
      unfolding nn_list_to_matrix_def
      by simp
    next
    case (Cons a xs)
    then show ?case
      unfolding nn_list_to_matrix_def
      by(cases a, auto split:if_splits)
  qed
qed
qed

```

```

lemma valid_activation_tab_seq_L_imp_m:
  <valid_activation_tab_l (activation_tab N) ⇒ valid_activation_tab_m (activation_tab (nn_list_to_matrix N))>
  unfolding valid_activation_tab_l_def valid_activation_tab_m_def nn_list_to_matrix_def activation_list_to_matrix_def
  o_def
  by (simp, smt (verit, del_insts) length_list_of_vec list_vec map_option_eq_Some mem_Collect_eq ran_def)

```

```

lemma layers_consistent_seq_L_imp_m:
  assumes <layers_consistent_l N n (layers N)>
  shows <layers_consistent_m (nn_list_to_matrix N) n (layers (nn_list_to_matrix N))>
proof(cases N)

```

```

case (fields layers activation_tab) note i = this
then show ?thesis
unfolding fields
proof(insert assms[simplified fields, simplified], induction layers arbitrary:n activation_tab)
case Nil
then show ?case
unfolding nn_list_to_matrix_def valid_activation_tab_l_def valid_activation_tab_m_def activation_list_to_matrix_def
o_def
by simp next
case (Cons a layers)
then show ?case proof(cases a)
case (In x1) note iii = this
then show ?thesis using Cons
unfolding In nn_list_to_matrix_def valid_activation_tab_l_def valid_activation_tab_m_def activa-
tion_list_to_matrix_def o_def
apply(simp add: Cons iii nn_list_to_matrix_def activation_list_to_matrix_def o_def)
using iii Cons layers_consistent_l_activation_tab_const
layers_consistent_m_activation_tab_const neural_network_seq_layers.select_convs(2)
by metis
next
case (Out x2)
then show ?thesis using Cons
unfolding nn_list_to_matrix_def valid_activation_tab_l_def valid_activation_tab_m_def activation_list_to_matrix_def
o_def
by (simp, metis (mono_tags, lifting) layers_consistent_l_activation_tab_const
layers_consistent_m_activation_tab_const neural_network_seq_layers.select_convs(2))
next
case (Dense x3) note iii = this
then show ?thesis
proof(cases x3)
case (fields name units  $\varphi$   $\beta$   $\omega$ )
then show ?thesis using Cons
apply(simp add: nn_list_to_matrix_def, safe)
subgoal
unfolding nn_list_to_matrix_def nn_list_to_matrix_def
apply(simp add: i Cons iii dim_col_list dim_col_mat_of_row_list dim_row_mat_of_row_list
nn_list_to_matrix_def)
by (smt (verit, ccfv_SIG) activation_list_to_matrix_def list.set_sel(1) list.size(3) not_less_iff_gr_or_eq o_def
option.map(2))
subgoal
unfolding nn_list_to_matrix_def nn_list_to_matrix_def
apply(simp add: i Cons iii dim_col_list dim_col_mat_of_row_list dim_row_mat_of_row_list
nn_list_to_matrix_def)
by (metis layers_consistent_l_activation_tab_const layers_consistent_m_activation_tab_const
neural_network_seq_layers.select_convs(2))
done
qed
next
case (Activation x4) note iii = this
then show ?thesis using Cons
apply(simp add: nn_list_to_matrix_def)
unfolding nn_list_to_matrix_def nn_list_to_matrix_def
apply(simp add: i iii dim_col_list dim_col_mat_of_row_list dim_row_mat_of_row_list
nn_list_to_matrix_def)

```

```

    by (metis (mono_tags, lifting) activation_list_to_matrix_def layers_consistentl_activation_tab_const
        layers_consistentm_activation_tab_const neural_network_seq_layers.select_convs(2) o_def option.map(2))
  qed
qed
qed

```

```

lemma in_deg_seq_l_eq_m: <in_deg_NN N = (in_deg_NN (nn_list_to_matrix N))>
proof(cases layers N)
  case Nil
  then show ?thesis
    unfolding in_deg_NN_def nn_list_to_matrix_def by simp
  next
  case (Cons a list)
  then show ?thesis
    unfolding in_deg_NN_def nn_list_to_matrix_def o_def
    by(cases a, auto split:if_splits)
qed

```

```

lemma is_In_seq_m_eq_l:
  assumes <(layers N) ≠ []>
  shows <isIn (List.hd (layers N)) = isIn (List.hd (layers (nn_matrix_to_list N)))>
proof(cases N)
  case (fields layers activation_tab)
  then show ?thesis
  proof(insert assms, cases layers)
    case Nil
    then show ?thesis using assms unfolding fields by simp
  next
  case (Cons a list)
  then show ?thesis
    using assms fields
    unfolding nn_matrix_to_list_def
    by(cases a, simp_all)
  qed
qed

```

```

lemma is_Out_seq_m_eq_l:
  assumes <(layers N) ≠ []>
  shows <isOut (last (layers N)) = isOut (last (layers (nn_matrix_to_list N)))>
proof(cases N)
  case (fields layers activation_tab) note i = this
  then show ?thesis unfolding fields
  proof(insert assms[simplified fields], induction layers)
    case Nil
    then show ?case by simp
  next
  case (Cons a layers)
  then show ?case
    unfolding nn_matrix_to_list_def
    by(cases a, simp_all)
  qed
qed

```

```

lemma is_Internal_seq_m_eq_l:

```

```

assumes <(layers N) ≠ []>
shows <list_all isInternal ((List.tl o butlast) (layers N)) = list_all isInternal ((List.tl o butlast) (layers (nn_matrix_to_list N)))>
proof(cases N)
case (fields layers activation_tab) note i = this
then show ?thesis unfolding fields
proof(insert assms[simplified fields], induction layers)
  case Nil
  then show ?case using Cons unfolding nn_matrix_to_list_def by simp
next
case (Cons x xs)
then show ?case
proof(induction xs)
  case Nil
  then show ?case using Cons unfolding nn_matrix_to_list_def by simp
next
case (Cons a xs)
then show ?case using Cons unfolding nn_matrix_to_list_def by (cases a, auto split:if_splits)
qed
qed
qed

```

```

lemma valid_activation_tab_seq_m_imp_l:
  <valid_activation_tab_m (activation_tab N) ⇒ valid_activation_tab_l (activation_tab (nn_matrix_to_list N))>
  unfolding valid_activation_tab_l_def valid_activation_tab_m_def nn_matrix_to_list_def activation_matrix_to_list_def
  o_def
  by (simp, smt (verit, del_insts) length_list_of_vec list_vec map_option_eq_Some mem_Collect_eq ran_def)

```

```

lemma layers_consistent_seq_m_imp_l:
assumes <layers_consistent_m N n (layers N)>
shows < layers_consistent_l (nn_matrix_to_list N) n (layers (nn_matrix_to_list N)) >
proof(cases N)
case (fields layers activation_tab) note i = this
then show ?thesis
  unfolding fields nn_matrix_to_list_def activation_matrix_to_list_def
proof(insert assms[simplified fields], induction layers arbitrary:n activation_tab)
  case Nil
  then show ?case
    unfolding nn_matrix_to_list_def valid_activation_tab_l_def valid_activation_tab_m_def activation_matrix_to_list_def
    o_def
    by simp next
    case (Cons a layers) note ii = this
    then show ?case proof(cases a)
      case (In x1) note iii = this
      then show ?thesis using Cons
        unfolding nn_matrix_to_list_def valid_activation_tab_l_def valid_activation_tab_m_def activation_matrix_to_list_def
        o_def
        by (simp, metis (mono_tags, lifting) layers_consistent_l_activation_tab_const
          layers_consistent_m_activation_tab_const neural_network_seq_layers.select_convs(2))
    next
    case (Out x2)
    then show ?thesis using Cons
      unfolding nn_matrix_to_list_def valid_activation_tab_l_def valid_activation_tab_m_def activation_matrix_to_list_def
      o_def

```

```

    by (simp, metis (mono_tags, lifting) layers_consistentl_activation_tab_const
        layers_consistentm_activation_tab_const neural_network_seq_layers.select_convs(2))
next
case (Dense x3) note iii = this
then show ?thesis
proof(cases x3)
case (fields name units  $\varphi \beta \omega$ )
then show ?thesis using Cons
apply(simp add: nn_matrix_to_list_def, safe)
subgoal
unfolding nn_matrix_to_list_def nn_matrix_to_list_def
apply(simp add: i ii iii dim_col_list dim_col_mat_of_row_list dim_row_mat_of_row_list
    nn_matrix_to_list_def)
by (metis dim_row_list index_transpose_mat(2))
subgoal
unfolding nn_matrix_to_list_def nn_matrix_to_list_def
apply(simp add: i ii iii dim_col_list dim_col_mat_of_row_list dim_row_mat_of_row_list
    nn_matrix_to_list_def)
by (metis layers_consistentl_activation_tab_const layers_consistentm_activation_tab_const
    neural_network_seq_layers.select_convs(2))
done
qed
next
case (Activation x4) note iii = this
then show ?thesis using Cons
apply(simp add: nn_matrix_to_list_def)
unfolding nn_matrix_to_list_def nn_matrix_to_list_def
apply(simp add: i ii iii dim_col_list dim_col_mat_of_row_list dim_row_mat_of_row_list
    nn_matrix_to_list_def)
by (metis layers_consistentl_activation_tab_const layers_consistentm_activation_tab_const
    neural_network_seq_layers.select_convs(2))
qed
qed
qed

lemma in_deg_seq_m_eq_l:  $\langle \text{in\_deg\_NN } N = (\text{in\_deg\_NN } (\text{nn\_matrix\_to\_list } N)) \rangle$ 
proof(cases layers N)
case Nil
then show ?thesis
unfolding in_deg_NN_def nn_matrix_to_list_def by simp
next
case (Cons a list)
then show ?thesis
unfolding in_deg_NN_def nn_matrix_to_list_def o_def
by(cases a, auto split:if_splits)
qed

theorem neural_network_sequential_L_m:
 $\langle \text{neural\_network\_sequential\_layers}_l N \implies \text{neural\_network\_sequential\_layers}_m (\text{nn\_list\_to\_matrix } N) \rangle$ 
unfolding neural_network_sequential_layersl_def neural_network_sequential_layersm_def
apply(safe)[1]
subgoal by (metis hd_Nil_eq_last isIn.elims(2) isOut.simps(2) is_In_seq_L_eq_m)
subgoal by (metis hd_Nil_eq_last isIn.elims(2) isOut.simps(2) is_Out_seq_L_eq_m)
subgoal by (metis hd_Nil_eq_last isIn.elims(2) isOut.simps(2) is_Internal_seq_L_eq_m)

```

subgoal using `valid_activation_tab_seq_l_imp_m` **by** `blast`
subgoal using `layers_consistent_seq_l_imp_m in_deg_seq_l_eq_m` **by** `metis`
done

theorem `neural_network_sequential_m_l`:
 $\langle \text{neural_network_sequential_layers}_m N \implies \text{neural_network_sequential_layers}_l (\text{nn_matrix_to_list } N) \rangle$
unfolding `neural_network_sequential_layers_l_def neural_network_sequential_layers_m_def`
apply(`safe`)[1]
subgoal by (`metis hd_Nil_eq_last isIn.elims(2) isOut.simps(2) is_In_seq_m_eq_l`)
subgoal by (`metis hd_Nil_eq_last isIn.elims(2) isOut.simps(2) is_Out_seq_m_eq_l`)
subgoal by (`metis hd_Nil_eq_last isIn.elims(2) isOut.simps(2) is_Internal_seq_m_eq_l`)
subgoal using `valid_activation_tab_seq_m_imp_l` **by** `blast`
subgoal using `layers_consistent_seq_m_imp_l in_deg_seq_m_eq_l` **by** `metis`
done

lemma `matrix_list_inverse`:
assumes $\langle \text{layers_consistent}_l N n (\text{layers } N) \rangle$
shows $\langle \text{nn_matrix_to_list } (\text{nn_list_to_matrix } N) = N \rangle$
proof(`cases N`)
case (`fields layers activation_tab`) **note** `i = this`
then show `?thesis`
unfolding `nn_matrix_to_list_def nn_list_to_matrix_def`
apply(`simp add: o_def activation_list_inverse'`)
using `assms layers_consistent_l All layer_list_matrix_inverse`
by (`metis (no_types, lifting) list.map_ident_strong neural_network_seq_layers.select_convs(1)`)
qed

lemma `list_matrix_inverse`:
assumes $\langle \text{layers_consistent}_m N n (\text{layers } N) \rangle$
shows $\langle \text{nn_list_to_matrix } (\text{nn_matrix_to_list } N) = N \rangle$
proof(`cases N`)
case (`fields layers activation_tab`) **note** `i = this`
then show `?thesis`
unfolding `nn_matrix_to_list_def nn_list_to_matrix_def`
apply(`simp add: o_def activation_matrix_inverse'`)
using `assms layers_consistent_m All`
by (`metis (no_types, lifting) layer_list_list_inverse list.map_ident_strong neural_network_seq_layers.select_convs(1)`)
qed

lemma `square_nth_nth_id`:
 $\forall w \in \text{set } ws. \text{length } w = \text{length } ws \implies$
 $(\text{map } (\lambda i. (\text{map } (\lambda ia. ws ! i ! ia) [0..<\text{length } ws]))) [0..<\text{length } ws] = ws$
by (`smt (verit, del_insts) in_set_conv_nth length_map map_cong map_nth nth_map`)

lemma `nth_map_f`: $\langle \text{map } ((\lambda i. f (xs ! i))) [0..<\text{length } xs] = \text{map } f \text{ xs} \rangle$
by (`smt (verit) add_o diff_zero length_map map_upt_eq_l nth_map`)

lemma `square_nth_nth_id_f`:
 $\forall w \in \text{set } ws. \text{length } w = \text{length } ws \implies$
 $(\text{map } (\lambda i. (\text{map } (\lambda ia. f (ws ! i ! ia) [0..<\text{length } ws]))) [0..<\text{length } ws] = \text{map } (\text{map } f) \text{ ws}$

```

using add_o diff_zero length_map map_upt_eq1 nth_map map_eq_conv map_nth nth_map_f[of map f ws]
square_nth_nth_id[of ws]
by(smt)

```

```

lemma F:⟨ length (ws::'a::{comm_ring} list) = length Inputs ⟹ map (λia. ws ! ia * Inputs ! ia) [0..<length Inputs] =
map2 (*) Inputs ws
by (simp add: map_equality_iff mult.commute)

```

```

lemma list_singleton: ⟨ length xs = 1 ⟹ ∃ e. xs = [e] ⟩
by (simp add: length_Suc_conv)

```

```

lemma activation_list_to_matrix_eq:
⟨ predictlayer_l N (Some (vs::'a::{comm_ring} list)) (Activation pl) =
map_option list_of_vec (predictlayer_m (nn_list_to_matrix N) (Some (vec_of_list vs))) ((layer_list_to_matrix
(Activation pl)))) ⟩
unfolding nn_list_to_matrix_def activation_list_to_matrix_def
by(auto simp add: vec_list list_vec split:option.split)

```

```

lemma layers_matrix_to_list:
⟨ (layers (nn_matrix_to_list N)) = map layer_matrix_to_list (layers N) ⟩
unfolding nn_matrix_to_list_def
by(simp)

```

```

lemma layers_list_to_matrix:
⟨ (layers (nn_list_to_matrix N)) = map layer_list_to_matrix (layers N) ⟩
unfolding nn_list_to_matrix_def
by(simp)

```

```

lemma layers_list_to_matrix':
⟨ layers N = l # ls ⟹ (layers (nn_list_to_matrix N)) = (layer_list_to_matrix l) # (map layer_list_to_matrix ls) ⟩
unfolding nn_list_to_matrix_def
by(simp)

```

```

lemma layers_list_to_matrix'':
⟨ (layers (nn_list_to_matrix (layers = l # ls, activation_tab = a))) = ((layer_list_to_matrix l) # (map
layer_list_to_matrix ls)) ⟩
by (simp add: layers_list_to_matrix')

```

```

lemma layers_list_to_matrix_none:
⟨ activation_tab N p = None ⟹ (activation_tab (nn_list_to_matrix N)) p = None ⟩
unfolding nn_list_to_matrix_def activation_list_to_matrix_def o_def
by(simp)

```

```

lemma layers_list_to_matrix_some:
⟨ activation_tab N p = Some f ⟹ (activation_tab (nn_list_to_matrix N)) p = Some (λx. vec_of_list (f (list_of_vec x))) ⟩
unfolding nn_list_to_matrix_def activation_list_to_matrix_def o_def
by(simp)

```

```

lemma activation_list_to_matrix:
⟨ (activation_tab (nn_list_to_matrix N)) = (activation_list_to_matrix (activation_tab N)) ⟩
unfolding nn_list_to_matrix_def activation_list_to_matrix_def o_def
by(simp)

```

lemma *vec_add_list*:
assumes $\langle \text{dim_vec } M = \text{length } bs \rangle$
shows $\langle M + \text{vec_of_list } bs = \text{vec_of_list } (\text{map2 } (+) (\text{list_of_vec } M) bs) \rangle$
using *assms unfolding plus_vec_def*
apply (*simp*)
by (*smt (verit, del_insts) dim_vec dim_vec_of_list eq_vec1 index_add_vec(1) index_add_vec(2) index_vec vec_add_list' vec_list vec_of_list_map*)

lemma *vec_add_list'*:
assumes $\langle \text{dim_vec } M = \text{dim_vec } bs \rangle$
shows $\langle M + bs = \text{vec_of_list } (\text{map2 } (+) (\text{list_of_vec } M) (\text{list_of_vec } bs)) \rangle$
using *assms unfolding plus_vec_def*
apply (*simp*)
by (*smt (verit, del_insts) dim_vec dim_vec_of_list eq_vec1 index_add_vec(1) index_add_vec(2) index_vec vec_add_list' vec_list vec_of_list_map*)

lemma *list_of_vec_map'*:
 $\langle v = \text{vec_of_list } (\text{map } ((\text{vec_index}) v) [0..<\text{dim_vec } v]) \rangle$
by (*metis list_of_vec_map vec_list*)

lemma *mat_list_transpose*:
assumes $\langle o < \text{dim_row } M \rangle$ **and** $\langle o < \text{dim_col } M \rangle$
shows $\langle (\text{mat_to_list } M^T) = \text{List.transpose } (\text{mat_to_list } M) \rangle$
using *assms*
unfolding *transpose_mat_def mat_to_list_def*
apply (*simp*)
unfolding *index_mat_def o_def map_fun_def id_def mat_def mk_mat_def*
apply (*subst Abs_mat_inverse*)
unfolding *mk_mat_def* **apply** (*blast*)
by (*subst transpose_rectangle, auto*)

lemma *dim_row_mat_not_zero*:
assumes $\langle \text{dim_row } M \neq 0 \rangle$
shows $\langle \text{mat_to_list } M \neq [] \rangle$
by (*metis assms dim_row_list list.size(3)*)

lemma *map2_to_map_idx_eq*: $\langle \text{length } xs = \text{length } ys \implies (\text{map2 } (*) xs ys) = \text{map } (\lambda i. xs!i * ys!i) [0..<\text{length } xs] \rangle$
using *map2_map_map map_nth*
by *metis*

lemma *map2_to_map_idx*: $\langle (\text{map2 } (*) xs ys) = \text{map } (\lambda i. xs!i * ys!i) [0..<\min (\text{length } xs) (\text{length } ys)] \rangle$
by (*rule nth_equality1, auto*)

lemma *length_list_transpose_mat*: $\langle o < \text{dim_row } M \implies o < \text{dim_col } M \implies \text{length } (\text{List.transpose } (\text{mat_to_list } M)) = \text{dim_col } M \rangle$
apply (*simp only: mat_list_transpose[symmetric] dim_row_list[symmetric]*)
by *simp*

lemma *map_sum_list_idx*: $\langle \text{map } (\lambda m. \text{sum_list } (\text{map2 } (*) m (\text{list_of_vec } v))) (\text{List.transpose } (\text{mat_to_list } M)) = \text{map } (\lambda i. \text{sum_list } (\text{map2 } (*) ((\text{List.transpose } (\text{mat_to_list } M))!i) (\text{list_of_vec } v))) [0..<\text{length } (\text{List.transpose } (\text{mat_to_list } M))] \rangle$

by (smt (verit, best) map_cong nth_map_f)

lemma *vec_mult_mat_list*:

assumes $\langle \forall as \in \text{set } (\text{mat_to_list } M). \text{length } as = \text{dim_col } M \rangle$

and $\langle \text{dim_vec } v = \text{dim_row } M \rangle$

and $\langle \text{dim_col } M \neq 0 \rangle$

and $\langle \text{dim_row } M \neq 0 \rangle$

shows $\langle (v :: 'a :: \text{comm_ring } \text{vec}) \ v * M = \text{vec_of_list } (\text{map } (\lambda m. \text{sum_list } (\text{map2 } (*) m (\text{list_of_vec } v))) (\text{mat_to_list } M^T)) \rangle$

apply(insert dim_row_mat_not_zero[of M])

apply(rule list_of_vec_ext)

apply(subst vec_list[of v, symmetric])

apply(subst list_mat[of M, symmetric])

unfolding mult_vec_mat_def scalar_prod_def

apply(simp only:vec_of_list_index)

apply(simp only:vec_list)

apply(subst col_of_rows_list')

using assms **apply**(simp)

using assms **apply**(simp)

unfolding sum_def

apply(subst comm_monoid_list_set.distinct_set_conv_list[of (+) o [0..<dim_vec v], simplified])

using sum.comm_monoid_list_set_axioms **apply** blast

apply(simp only:sum_list_def[symmetric])

apply(simp only:list_vec)

apply(simp only:vec_of_list_index)

apply(simp add:dim_col_mat_of_row_list)

using assms dim_row_list length_o_conv

apply(simp only:map_sum_list_idx)

apply(subst map2_to_map_idx)

apply (rule nth_equality)

apply (simp add: mat_to_list_def)

apply(simp only: mat_list_transpose)

using assms length_list_transpose_mat[of M] nth_transpose[of _ (mat_to_list M), simplified]

apply(simp)

by (simp add: dim_row_list)

lemma *hd_length_inputs*: $\langle 0 < \text{units } x3 \implies$

$\text{length } (\beta x3) = \text{units } x3 \implies \text{length } (\omega x3) = \text{units } x3 \implies \forall r \in \text{set } (\omega x3). \text{length } r = \text{length } \text{Inputs} \implies \text{length } \text{Inputs} = \text{length } (\text{List.hd } (\omega x3)) \rangle$

by (metis length_greater_o_conv list.set_sel(1))

5.3.3 Semantic Equivalence of List-based and Matrix-based Models

lemma *ln_l_to_m_eq*:

$\langle \text{predict}_{\text{layer_l}} N (\text{Some } vs) (\text{In } l) = \text{map_option } \text{list_of_vec } (\text{predict}_{\text{layer_m}} (\text{nn_list_to_matrix } N) (\text{Some } (\text{vec_of_list } vs))) (\text{layer_list_to_matrix } (\text{In } l)) \rangle$

by(simp add:list_vec)

lemma *ln_l_to_m_eq'*:

$\langle \text{predict}_{\text{layer_m}} (\text{nn_list_to_matrix } N) (\text{Some } (\text{vec_of_list } vs)) (\text{layer_list_to_matrix } (\text{In } l)) = \text{map_option } \text{vec_of_list } (\text{predict}_{\text{layer_l}} N (\text{Some } vs) (\text{In } l)) \rangle$

by(simp add:list_vec)

lemma Out_l_to_m_eq:

⟨predict_{layer_l} N (Some vs) (Out l) = map_option list_of_vec (predict_{layer_m} (nn_list_to_matrix N) (Some (vec_of_list vs))) (layer_list_to_matrix (Out l))⟩

by(simp add:list_vec)

lemma Out_l_to_m_eq':

⟨predict_{layer_m} (nn_list_to_matrix N) (Some (vec_of_list vs)) (layer_list_to_matrix (Out l)) = map_option vec_of_list (predict_{layer_l} N (Some vs) (Out l))⟩

by(simp add:list_vec)

lemma Dense_l_to_m_eq:

assumes ⟨layer_consistent_l N (length vs) (Dense l)⟩

shows ⟨predict_{layer_l} N (Some (vs::'a::comm_ring list)) (Dense l)

= map_option list_of_vec (predict_{layer_m} (nn_list_to_matrix N) (Some (vec_of_list vs)) (layer_list_to_matrix (Dense l)))⟩

proof(cases activation_tab N (φ l) = None)

case True

then show ?thesis **by**(auto simp add: nn_list_to_matrix_def activation_list_to_matrix_def o_def)

next

case False

then show ?thesis

using assms

apply(simp add: nn_list_to_matrix_def activation_list_to_matrix_def o_def)[1]

apply (simp add: dim_mult_vec_mat dim_row_mat_of_row_list)

apply(subst vec_mult_mat_list)

subgoal **by**(simp add: dim_col_list)

subgoal **by** (simp add: dim_col_mat_of_row_list hd_length_inputs)

subgoal **by**(simp add: dim_row_mat_of_row_list)

subgoal **by**(simp add: dim_col_mat_of_row_list, metis gr_implies_noto hd_in_set length_o_conv)

subgoal **apply**(clarsimp simp add:list_vec)

subgoal **apply**(subst mat_list)

subgoal **by**(metis length_greater_o_conv list.set_sel(1))

subgoal **using** list_vec map2_mult_commute map_eq_conv vec_of_list_map

apply(clarsimp simp add:o_def)

using Matrix_Utils.vec_add_list length_map map2_mult_commute map_eq_conv vec_of_list_map

by (smt (verit) dim_col_mat_of_row_list hd_length_inputs)

done

done

done

qed

lemma Dense_l_to_m_eq':

assumes ⟨layer_consistent_l N (length vs) (Dense l)⟩

shows ⟨predict_{layer_m} (nn_list_to_matrix N) (Some (vec_of_list vs)) (layer_list_to_matrix (Dense l))

= map_option vec_of_list (predict_{layer_l} N (Some (vs::'a::comm_ring list)) (Dense l))⟩

using Dense_l_to_m_eq

by (smt (verit) assms not_Some_eq option.simps(8) option.simps(9) vec_list)

lemma Activation_l_to_m_eq:

```

⟨predictlayer_l N (Some vs) (Activation l)
= map_option list_of_vec (predictlayer_m (nn_list_to_matrix N) (Some (vec_of_list vs)) (layer_list_to_matrix
(Activation l)))⟩
proof(cases activation_tab N (φ l) = None)
case True
then show ?thesis by(auto simp add: nn_list_to_matrix_def activation_list_to_matrix_def o_def )
next
case False
then show ?thesis
apply(simp add:list_vec nn_list_to_matrix_def activation_list_to_matrix_def)
by(simp add: vec_list list_vec split:option.splits)
qed

```

lemma Activation_l_to_m_eq':

```

⟨predictlayer_m (nn_list_to_matrix N) (Some (vec_of_list vs)) (layer_list_to_matrix (Activation l))
= map_option vec_of_list (predictlayer_l N (Some vs) (Activation l))⟩
by (smt (verit, del_insts) Activation_l_to_m_eq option.exhaust_sel option.map_disc_iff option.map_sel vec_list)

```

lemma aux1: <

```

∧y. l = Dense x3 ⇒
  (∧Inputs.
    layers_consistentl (|layers = l0, activation_tab = activation_tab'|) (length Inputs) layers' ⇒
      foldl (predictlayer_l (|layers = l1, activation_tab = activation_tab'|)) (Some Inputs) layers' =
        map_option list_of_vec (foldl (predictlayer_m (nn_list_to_matrix (|layers = l2, activation_tab = activation_tab'|)))
        (Some (vec_of_list Inputs)) (layers (nn_list_to_matrix (|layers = layers', activation_tab = a2)))))) ⇒
      valid_activation_tabl activation_tab' ⇒
      0 < units x3 ⇒
      Inputs ≠ [] ⇒
      length (LayerRecord.β x3) = units x3 ⇒
      length (LayerRecord.ω x3) = units x3 ⇒
      ∀ r ∈ set (LayerRecord.ω x3). length r = length Inputs ⇒
      layers_consistentl (|layers = l0, activation_tab = activation_tab'|) (units x3) layers' ⇒
      activation_tab' (ActivationRecord.φ x3) = Some y ⇒
      foldl (predictlayer_l (|layers = l1, activation_tab = activation_tab'|)) (Some (y (map2 (+) (map ((λvs'. ∑ (x,
y) ← vs'. x * y) ∘ zip Inputs) (LayerRecord.ω x3)) (LayerRecord.β x3)))) layers' =
      map_option list_of_vec
      (foldl (predictlayer_m (nn_list_to_matrix (|layers = l2, activation_tab = activation_tab'|))) (Some (vec_of_list (y
(map2 (+) (map ((λvs'. ∑ (x, y) ← vs'. x * y) ∘ zip Inputs) (LayerRecord.ω x3)) (LayerRecord.β x3))))))
      (map layer_list_to_matrix layers')) >

```

proof —

```

fix y :: 'a list ⇒ 'a list
assume a1: valid_activation_tabl activation_tab'
assume a2: length (β x3) = units x3
assume a3: length (ω x3) = units x3
assume a4: activation_tab' (φ x3) = Some y
assume a5: layers_consistentl (|layers = l0, activation_tab = activation_tab'|) (units x3) layers'
assume a6: ∧Inputs. layers_consistentl (|layers = l0, activation_tab = activation_tab'|) (length Inputs) layers'
  ⇒ foldl (predictlayer_l (|layers = l1, activation_tab = activation_tab'|)) (Some Inputs) layers' = map_option
list_of_vec (foldl (predictlayer_m (nn_list_to_matrix (|layers = l2, activation_tab = activation_tab'|))) (Some
(vec_of_list Inputs)) (layers (nn_list_to_matrix (|layers = layers', activation_tab = a2))))
have ∧as. length (y as) = length as
using a4 a1 by (metis (no_types) NN_Layers_List_Main.valid_activation_preserves_length)
then show foldl (predictlayer_l (|layers = l1, activation_tab = activation_tab'|)) (Some (y (map2 (+) (map

```

```

((λps. ∑ (x, y) ← ps. x * y) ∘ zip Inputs) (ω x3) (β x3))) layers' = map_option list_of_vec (foldl (predict_layer_m
(nn_list_to_matrix (layers = l2, activation_tab = activation_tab'))) (Some (vec_of_list (y (map2 (+) (map ((λps. ∑ (x,
y) ← ps. x * y) ∘ zip Inputs) (ω x3) (β x3))))) (map layer_list_to_matrix layers')))
using a6 a5 a3 a2 by (simp add: nn_list_to_matrix_def)
qed

```

lemma *predict_seq_l_eq_m'*:

assumes $\langle \text{layers_consistent}_l \ (\! \text{layers} = l_0, \text{activation_tab} = \text{activation_tab}' \!) \ (\text{length} \ (\text{Inputs}::'a::\text{comm_ring list})) \ \text{layers}' \rangle$

and $\langle \text{valid_activation_tab}_l \ \text{activation_tab}' \rangle$

shows $\langle \text{foldl} \ (\text{predict}_{\text{layer}_l} \ (\! \text{layers} = l_1, \text{activation_tab} = \text{activation_tab}' \!)) \ (\text{Some} \ (\text{Inputs})) \ (\text{layers} \ (\! \text{layers} = \text{layers}', \text{activation_tab} = a_1 \!)) =$

$\text{map_option list_of_vec}$

$(\text{foldl} \ (\text{predict}_{\text{layer}_m} \ (\! \text{layers} = l_2, \text{activation_tab} = \text{activation_tab}' \!)) \ (\text{Some} \ (\text{vec_of_list Inputs}))$

$(\text{layers} \ (\! \text{layers} = \text{layers}', \text{activation_tab} = a_2 \!)) \rangle$

proof(*insert assms, induction layers' arbitrary: Inputs*)

case Nil *then show ?case*

unfolding *predict_seq_layer_l_def predict_seq_layer_m'_def predict_seq_layer_m_def*
nn_list_to_matrix_def o_def activation_list_to_matrix_def in_deg_NN_def

by(*simp add:list_vec*)

next

case (*Cons l layers'*)

then show ?case

proof(*cases l*)

case (*In x1*)

then show ?thesis

apply(*simp add:Cons*)

using *Cons*

by (*simp add: layers_list_to_matrix list_vec fold_predict_l_strict fold_predict_m_strict*)

next

case (*Out x2*)

then show ?thesis

using *Cons*

by (*simp add: layers_list_to_matrix list_vec fold_predict_l_strict fold_predict_m_strict*)

next

case (*Dense x3*) *note i = this*

then show ?thesis

apply(*predict_layer add:layers_list_to_matrix*)

apply(*subst Dense_l_to_m_eq'*)

using *Cons.premis(1) layers_consistent_l.simps(2) apply(simp)*

using *Cons.IH Cons.premis(1) assms(2)*

apply(*simp*)

using *aux1*

by (*smt (verit) FNN_Layers_List_Main.valid_activation_preserves_length layers_list_to_matrix length_map length_upt list.map_comp*

neural_network_seq_layers.simps(1) option.simps(5,9) verit_minus_simplify(2))

next

case (*Activation x4*)

then show ?thesis

apply(*predict_layer add:layers_list_to_matrix*)

apply(*subst Activation_l_to_m_eq'*)

using *Cons.IH Cons.premis(1) assms(2)*

apply(*simp*)

```

by (metis (mono_tags, lifting) NN_Layers_List_Main.valid_activation_preserves_length layers_list_to_matrix
neural_network_seq_layers.select_convs(1) option.simps(5,9))
qed
qed

```

```

theorem predict_seq_l_eq_m:
  assumes <layers_consistentl N (length Inputs) (layers N)>
  and <valid_activation_tabl (activation_tab N)>
  shows <predictseq_layer_l N (Inputs::'a::comm_ring list) = predictseq_layer_m' (nn_list_to_matrix N) Inputs>
proof(cases N)
case (fields layers activation_tab)
then show ?thesis
  using assms
  apply(simp)
  unfolding predictseq_layer_l_def predictseq_layer_m'_def predictseq_layer_m_def
  by(subst predict_seq_l_eq_m', simp_all)
qed

```

```

lemma In_m_to_l_eq:
  <predictlayer_m N (Some vs) (In l) = map_option vec_of_list (predictlayer_l (nn_matrix_to_list N) (Some (list_of_vec
vs))) (layer_matrix_to_list (In l))>
  by(simp add:vec_list)

```

```

lemma In_m_to_l_eq':
  <predictlayer_l (nn_matrix_to_list N) (Some (list_of_vec vs)) (layer_matrix_to_list (In l)) = map_option list_of_vec
(predictlayer_m N (Some vs) (In l)) >
  by(simp add:vec_list)

```

```

lemma Out_m_to_l_eq:
  <predictlayer_m N (Some vs) (Out l) = map_option vec_of_list (predictlayer_l (nn_matrix_to_list N) (Some (list_of_vec
vs))) (layer_matrix_to_list (Out l))>
  by(simp add:vec_list)

```

```

lemma Out_m_to_l_eq':
  <predictlayer_l (nn_matrix_to_list N) (Some (list_of_vec vs)) (layer_matrix_to_list (In l)) = map_option list_of_vec
(predictlayer_m N (Some vs) (Out l)) >
  by(simp add:vec_list)

```

```

lemma Dense_m_to_l_eq:
  assumes <layer_consistentm N (dim_vec vs) (Dense l)>
  shows <predictlayer_m N (Some (vs::'a::comm_ring Matrix.vec)) (Dense l)
  = map_option vec_of_list (predictlayer_l (nn_matrix_to_list N) (Some (list_of_vec vs))) (layer_matrix_to_list (Dense
l))>
proof(cases activation_tab N (φ l) = None)
case True
then show ?thesis by(auto simp add: nn_matrix_to_list_def activation_matrix_to_list_def o_def )
next
case False
then show ?thesis
  apply(clarsimp simp add: nn_matrix_to_list_def activation_matrix_to_list_def o_def)[1]
  subgoal using assms
  apply(simp add: nn_matrix_to_list_def activation_matrix_to_list_def o_def)[1]
  apply(subst vec_add_list')

```

```

    apply (simp add: dim_mult_vec_mat dim_row_mat_of_row_list)
  apply (simp)
  apply (subst vec_mult_mat_list)
  subgoal by (simp add: dim_col_list)
  subgoal by (simp add: dim_col_mat_of_row_list hd_length_inputs)
  subgoal by (simp add: dim_row_mat_of_row_list)
  subgoal by (auto simp add: dim_col_mat_of_row_list)
  subgoal apply (simp add: list_vec map2_mult_commute vec_list list_mat_transpose_transpose)
  by (smt (z3) Matrix_Utils.vec_add_list dim_col_mat_list dim_vec_of_list index_map_vec(2) index_transpose_mat(3)
    length_list_transpose_mat mat_list_transpose nat_neq_iff vec_list vec_of_dim_o vec_of_list_map zero_vec_zero)
  done
done
qed

```

```

lemma Dense_m_to_L_eq':
  assumes ⟨layer_consistentm N (dim_vec vs) (Dense l)⟩
  shows ⟨predictlayer_l (nn_matrix_to_list N) (Some (list_of_vec vs)) (layer_matrix_to_list (Dense l))
    = map_option list_of_vec (predictlayer_m N (Some (vs::'a::comm_ring Matrix.vec))) (Dense l)⟩
  using Dense_m_to_L_eq
  by (smt (verit) assms not_Some_eq option.simps(8) option.simps(9) list_vec)

```

```

lemma Activation_m_to_L_eq:
  ⟨predictlayer_m N (Some vs) (Activation l)
  = map_option vec_of_list (predictlayer_l (nn_matrix_to_list N) (Some (list_of_vec vs)) (layer_matrix_to_list (Activation l)))⟩
proof (cases activation_tab N (φ l) = None)
  case True
  then show ?thesis by (auto simp add: nn_matrix_to_list_def activation_matrix_to_list_def o_def)
next
  case False
  then show ?thesis
  apply (simp add: list_vec nn_matrix_to_list_def activation_matrix_to_list_def)
  by (simp add: vec_list list_vec split:option.splits)
qed

```

```

lemma Activation_m_to_L_eq':
  ⟨predictlayer_l (nn_matrix_to_list N) (Some (list_of_vec vs)) (layer_matrix_to_list (Activation l))
  = map_option list_of_vec (predictlayer_m N (Some vs) (Activation l))⟩
  by (smt (verit, del_insts) Activation_m_to_L_eq option.exhaust_sel option.map_disc_iff option.map_sel list_vec)

```

```

lemma predict_seq_m_eq_l':
  assumes ⟨layers_consistentm (⟦layers = lo, activation_tab = activation_tab'⟧) (dim_vec (Inputs::'a::comm_ring Matrix.vec)) layers'⟩
  and ⟨valid_activation_tabm activation_tab'⟩
  shows ⟨foldl (predictlayer_m (⟦layers = l1, activation_tab = activation_tab'⟧)) (Some (Inputs)) (layers (⟦layers = layers',
activation_tab = a1⟧)) =
  map_option vec_of_list
  (foldl (predictlayer_l (nn_matrix_to_list (⟦layers = l2, activation_tab = activation_tab'⟧)) (Some (list_of_vec Inputs))
  (layers (nn_matrix_to_list (⟦layers = layers', activation_tab = a2⟧))))⟩
proof (insert assms, induction layers' arbitrary: Inputs a)
  case Nil then show ?case
  unfolding predictseq_layer_l_def predictseq_layer_m'_def predictseq_layer_m_def
  nn_matrix_to_list_def o_def activation_matrix_to_list_def in_deg_NN_def
  by (simp add: vec_list)

```

```

next
case (Cons l layers^)
then show ?case
proof(cases l)
case (In x1)
then show ?thesis
using Cons
by (auto simp add: layers_matrix_to_list vec_list fold_predict_l_strict fold_predict_m_strict)
next
case (Out x2)
then show ?thesis
using Cons
by (auto simp add: layers_matrix_to_list vec_list fold_predict_l_strict fold_predict_m_strict)
next
case (Dense x3) note i = this
then show ?thesis
apply(predict_layer add:layers_matrix_to_list)
apply(subst Dense_m_to_l_eq')
using Cons.prem1 layers_consistent_m.simp2 apply(simp)
using Cons.IH Cons.prem1 assms2
apply(clarsimp)[1]
by (metis (no_types, lifting) index_add_vec2 layers_matrix_to_list
neural_network_seq_layers.select_convs1 valid_activation_preserves_dim)
next
case (Activation x4)
then show ?thesis
apply(predict_layer add:layers_matrix_to_list)
apply(subst Activation_m_to_l_eq')
using Cons.IH Cons.prem1 assms2
apply(clarsimp)[1]
by (metis (no_types, lifting) layers_matrix_to_list neural_network_seq_layers.select_convs1
valid_activation_preserves_dim)
qed
qed

theorem predict_seq_m_eq_l:
assumes ⟨layers_consistent_m N (length Inputs) (layers N)⟩
and ⟨valid_activation_tab_m (activation_tab N)⟩
shows ⟨predict_seq_layer_m' N (Inputs::'a::comm_ring list) = predict_seq_layer_l (nn_matrix_to_list N) Inputs⟩
proof(cases N)
case (fields layers activation_tab)
then show ?thesis
using assms
apply(simp)
unfolding predict_seq_layer_l_def predict_seq_layer_m'_def predict_seq_layer_m_def
apply(subst predict_seq_m_eq_l', simp_all)
by (smt (verit, ccfv_threshold) assms1 layers_consistent_seq_m_imp_l layers_matrix_to_list
length_list_of_vec list_matrix_inverse list_vec neural_network_seq_layers.select_convs2
nn_matrix_to_list_def predict_seq_l_eq_m' predict_seq_m_eq_l' valid_activation_tab_seq_m_imp_l)
qed

corollary predict_seq_m_eq_l2:
assumes ⟨layers_consistent_m N (dim_vec Inputs) (layers N)⟩
and ⟨valid_activation_tab_m (activation_tab N)⟩

```

```
shows <map_option list_of_vec (predictseq_layer_m N Inputs) = predictseq_layer_l (nn_matrix_to_list N) (list_of_vec  
Inputs)>  
using precdict_seq_m_eq_l predictseq_layer_m ' _def dim_vec_of_list  
by (metis assms(1) assms(2) vec_list)  
end
```


6 Main Theory Including all Model Types (📄 NN_Main)

```
theory
  NN_Main
imports
  NN_Digraph_Main
  NN_Layers_Main
begin

end
```


7 Reference Manual (thy)

theory

 NN_Manual

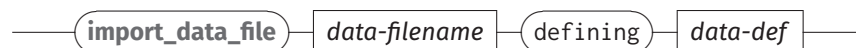
imports

 NN_Main

begin

7.1 Importing Neural Networks and Data (📄 NN_Manual)

import_data_file. For importing test or training data, we provide the following command:



where *data-filename* is the path (file name) of the data file that should be read and *data-def* is the name the HOL definition representing the data is bound to. The data file should be a simple two-dimensional array of real numbers as, e.g., produced by NumPy's [11] `saveetxt` command:

```
import numpy as np
training_data = np.array([[0,0],[0,1],[1,0],[1,1]], "float32")
np.savetxt('training_data.out', training_data)
```

Python

For further information, please see <https://numpy.org/doc/stable/reference/generated/numpy.savetxt.html>.

import_TensorFlow. For importing trained neural networks, we provide the following command:



The input should be in JSON [9] format with the weights stored in a separate file. This format is used by TensorFlow.js [14] and also supported by the corresponding Python module. For example:

```

import tensorflowjs as tfjs
import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense

training_data = np.array([[0,0],[0,1],[1,0],[1,1]], "float32")
target_data = np.array([[1],[0],[0],[0]], "float32")

model = Sequential()
model.add(Dense(1, activation='hard_sigmoid'))
model.compile(loss='mean_squared_error', optimizer='adam', metrics=['binary_accuracy'])
model.fit(training_data, target_data, epochs=7500, verbose=0)

# safe trained model as JSON (with external file for weights)
tfjs.converters.save_keras_model(model, ".")

```

Configuration. We provide several configuration attributes:

- The attribute *nn_proof_mode* (default *nbe* configures if proofs during the import of neural networks (i.e., *import_TensorFlow*) should
 - not generate any proofs (*skip*)
 - generate proofs axiomatically, without actually proving them (*sorry*)
 - use code generation (i.e., the proof method *eval*) if possible (*eval*)
 - use normalization by evaluation (i.e., the proof method *normalization*) if possible (*nbe*)
 - avoid using the code generator (*safe*)

While, in many scenarios, the proof method *eval* is much faster than the alternative approaches, its safety relies on the configuration of the code generator. A more detailed discussion can be found in Section 5 of [10].

7.2 Proof Methods (📖 NN_Manual)

Currently, we provide two domain-specific proof methods:

- The method *predict_layer* is, in its essence, a simplification using the theorem set *nn_layer*, which is configured automatically by the import of layer-based models.
- *forced_approximation* is a method mainly for debugging and experimentation that repeats the application of the *approximation*.

end

8 Examples

8.1 Compass

8.1.1 Neural Networks as Directed Graphs (Compass_Digraph)

```
theory
  Compass_Digraph
  imports
    Neural_Networks.NN_Digraph_Main
begin
```

Manual Encoding

Definition: Neurons `definition m_No` \equiv `In 0`

`definition m_N1` \equiv `In 1`

`definition m_N2` \equiv `In 2`

`definition m_N3` \equiv `In 3`

`definition m_N4` \equiv `In 4`

`definition m_N5` \equiv `In 5`

`definition m_N6` \equiv `In 6`

`definition m_N7` \equiv `In 7`

`definition m_N8` \equiv `In 8`

`definition m_N9` \equiv `Neuron` ($\varphi = \text{Identity}, \alpha = 1, \beta = 7082077 / 1000000000, \text{uid} = 9$)

`definition m_N10` \equiv `Neuron` ($\varphi = \text{Identity}, \alpha = 1, \beta = 107544795 / 1000000000, \text{uid} = 10$)

`definition m_N11` \equiv `Neuron` ($\varphi = \text{Identity}, \alpha = 1, \beta = -15743796 / 1000000000, \text{uid} = 11$)

`definition m_N12` \equiv `Neuron` ($\varphi = \text{Identity}, \alpha = 1, \beta = 47920802 / 1000000000, \text{uid} = 12$)

`definition m_N13` \equiv `Neuron` ($\varphi = \text{Identity}, \alpha = 1, \beta = -16364478 / 1000000000, \text{uid} = 13$)

`definition m_N14` \equiv `Neuron` ($\varphi = \text{Identity}, \alpha = 1, \beta = -24132763 / 1000000000, \text{uid} = 14$)

`definition m_N15` \equiv `Neuron` ($\varphi = \text{Identity}, \alpha = 1, \beta = -30579916 / 1000000000, \text{uid} = 15$)

`definition m_N16` \equiv `Out 16`

`definition m_N17` \equiv `Out 17`

`definition m_N18` \equiv `Out 18`

`definition m_N19` \equiv `Out 19`

lemmas

`m_neuron_defs` = `m_No_def m_N1_def m_N2_def m_N3_def m_N4_def m_N5_def m_N6_def m_N7_def m_N8_def`
`m_N9_def m_N10_def m_N11_def m_N12_def m_N13_def m_N14_def m_N15_def m_N16_def`
`m_N17_def m_N18_def m_N19_def`

`definition m_Neurons` = [`m_No, m_N1, m_N2, m_N3, m_N4, m_N5, m_N6, m_N7, m_N8, m_N9, m_N10, m_N11,`
`m_N12, m_N13, m_N14, m_N15, m_N16, m_N17, m_N18, m_N19`]

Definition: Edges `definition m_E12_16` \equiv ($\omega = 1, \text{tl} = \text{m_N12}, \text{hd} = \text{m_N16}$)

`definition m_E13_17` \equiv ($\omega = 1, \text{tl} = \text{m_N13}, \text{hd} = \text{m_N17}$)

`definition m_E14_18` \equiv ($\omega = 1, \text{tl} = \text{m_N14}, \text{hd} = \text{m_N18}$)

`definition m_E15_19` \equiv ($\omega = 1, \text{tl} = \text{m_N15}, \text{hd} = \text{m_N19}$)

`definition m_E9_12` \equiv ($\omega = 8217673 / 200000000, \text{tl} = \text{m_N9}, \text{hd} = \text{m_N12}$)

definition $m_E10_12 \equiv (\omega = 2972081 / 20000000, tl = m_N10, hd = m_N12)$
definition $m_E11_12 \equiv (\omega = 2445593 / 10000000, tl = m_N11, hd = m_N12)$
definition $m_E9_13 \equiv (\omega = - (11993983 / 5000000), tl = m_N9, hd = m_N13)$
definition $m_E10_13 \equiv (\omega = - (3894687 / 50000000), tl = m_N10, hd = m_N13)$
definition $m_E11_13 \equiv (\omega = 646179 / 1250000, tl = m_N11, hd = m_N13)$
definition $m_E9_14 \equiv (\omega = - (2323241 / 5000000), tl = m_N9, hd = m_N14)$
definition $m_E10_14 \equiv (\omega = 10928257 / 10000000, tl = m_N10, hd = m_N14)$
definition $m_E11_14 \equiv (\omega = - (7042477 / 5000000), tl = m_N11, hd = m_N14)$
definition $m_E9_15 \equiv (\omega = 19465483 / 10000000, tl = m_N9, hd = m_N15)$
definition $m_E10_15 \equiv (\omega = - (9524061 / 10000000), tl = m_N10, hd = m_N15)$
definition $m_E11_15 \equiv (\omega = - (31743723 / 50000000), tl = m_N11, hd = m_N15)$
definition $m_E0_9 \equiv (\omega = 3342313 / 5000000, tl = m_N0, hd = m_N9)$
definition $m_E1_9 \equiv (\omega = - (12952799 / 10000000), tl = m_N1, hd = m_N9)$
definition $m_E2_9 \equiv (\omega = - (1428979 / 5000000), tl = m_N2, hd = m_N9)$
definition $m_E3_9 \equiv (\omega = 8650103 / 5000000, tl = m_N3, hd = m_N9)$
definition $m_E4_9 \equiv (\omega = 63918763 / 100000000, tl = m_N4, hd = m_N9)$
definition $m_E5_9 \equiv (\omega = - (6959659 / 5000000), tl = m_N5, hd = m_N9)$
definition $m_E6_9 \equiv (\omega = - (9054079 / 20000000), tl = m_N6, hd = m_N9)$
definition $m_E7_9 \equiv (\omega = 13654941 / 10000000, tl = m_N7, hd = m_N9)$
definition $m_E8_9 \equiv (\omega = - (18450487 / 100000000), tl = m_N8, hd = m_N9)$
definition $m_E0_10 \equiv (\omega = 314303 / 5000000, tl = m_N0, hd = m_N10)$
definition $m_E1_10 \equiv (\omega = 915709 / 2500000, tl = m_N1, hd = m_N10)$
definition $m_E2_10 \equiv (\omega = 6922799 / 10000000, tl = m_N2, hd = m_N10)$
definition $m_E3_10 \equiv (\omega = - (9399607 / 25000000), tl = m_N3, hd = m_N10)$
definition $m_E4_10 \equiv (\omega = 15055849 / 100000000, tl = m_N4, hd = m_N10)$
definition $m_E5_10 \equiv (\omega = 10981513 / 10000000, tl = m_N5, hd = m_N10)$
definition $m_E6_10 \equiv (\omega = 3420911 / 200000000, tl = m_N6, hd = m_N10)$
definition $m_E7_10 \equiv (\omega = 7420693 / 10000000, tl = m_N7, hd = m_N10)$
definition $m_E8_10 \equiv (\omega = 15639223 / 100000000, tl = m_N8, hd = m_N10)$
definition $m_E0_11 \equiv (\omega = 9863281 / 100000000, tl = m_N0, hd = m_N11)$
definition $m_E1_11 \equiv (\omega = 9530481 / 10000000, tl = m_N1, hd = m_N11)$
definition $m_E2_11 \equiv (\omega = 35006753 / 100000000, tl = m_N2, hd = m_N11)$
definition $m_E3_11 \equiv (\omega = 7897923 / 10000000, tl = m_N3, hd = m_N11)$
definition $m_E4_11 \equiv (\omega = - (11627171 / 20000000), tl = m_N4, hd = m_N11)$
definition $m_E5_11 \equiv (\omega = 2839861 / 5000000, tl = m_N5, hd = m_N11)$
definition $m_E6_11 \equiv (\omega = 5311743 / 10000000, tl = m_N6, hd = m_N11)$
definition $m_E7_11 \equiv (\omega = - (9090567 / 10000000), tl = m_N7, hd = m_N11)$
definition $m_E8_11 \equiv (\omega = - (181917 / 400000), tl = m_N8, hd = m_N11)$

lemmas

$m_edge_defs = m_E12_16_def m_E13_17_def m_E14_18_def m_E15_19_def m_E9_12_def m_E10_12_def$
 $m_E11_12_def m_E9_13_def m_E10_13_def m_E11_13_def m_E9_14_def m_E10_14_def$
 $m_E11_14_def m_E9_15_def m_E10_15_def m_E11_15_def m_E0_9_def m_E1_9_def m_E2_9_def$
 $m_E3_9_def m_E4_9_def m_E5_9_def m_E6_9_def m_E7_9_def m_E8_9_def m_E0_10_def$
 $m_E1_10_def m_E2_10_def m_E3_10_def m_E4_10_def m_E5_10_def m_E6_10_def m_E7_10_def$
 $m_E8_10_def m_E0_11_def m_E1_11_def m_E2_11_def m_E3_11_def m_E4_11_def m_E5_11_def$
 $m_E6_11_def m_E7_11_def m_E8_11_def$

definition

$\langle m_Edges = [m_E12_16, m_E13_17, m_E14_18, m_E15_19, m_E9_12, m_E10_12, m_E11_12, m_E9_13, m_E10_13,$
 $m_E11_13, m_E9_14, m_E10_14, m_E11_14, m_E9_15, m_E10_15, m_E11_15, m_E0_9, m_E1_9,$
 $m_E2_9, m_E3_9, m_E4_9, m_E5_9, m_E6_9, m_E7_9, m_E8_9, m_E0_10, m_E1_10, m_E2_10,$
 $m_E3_10, m_E4_10, m_E5_10, m_E6_10, m_E7_10, m_E8_10, m_E0_11, m_E1_11, m_E2_11,$
 $m_E3_11, m_E4_11, m_E5_11, m_E6_11, m_E7_11, m_E8_11] \rangle$

definition

```
⟨m_Graph ≡ mk_nn_pregraph m_Edges⟩
```

Definition: Activation Tab fun m_φ_compass where

```
⟨m_φ_compass Identity = Some identity⟩  
| ⟨m_φ_compass _ = None⟩
```

Definition: Neural Network definition

```
⟨m_NeuralNet = (graph = m_Graph, activation_tab = m_φ_compass)⟩
```

Locale Interpretations global_interpretation m_compass: nn_pregraph m_Graph

```
apply(simp add: m_Graph_def)  
apply(simp add: nn_pregraph_mk nn_pregraph.axioms)  
done
```

Automated Encoding Using The TensorFlow Import

```
Single Encoding declare [[nn_proof_mode = eval]]  
import_TensorFlow compass file model/model.json as digraph_single  
declare [[nn_proof_mode = nbe]]
```

```
thm compass.neuron_defs  
thm compass.Neurons_def  
thm compass.edge_defs  
thm compass.Edges_def  
thm compass.Graph_def  
thm compass.verts_set_conv  
thm compass.edges_set_conv  
thm compass.φ_compass.simps  
thm compass.NeuralNet_def  
thm compass.nn_pregraph_axioms  
thm compass.neural_network_digraph_single_axioms
```

importing the data files

```
import_data_file model/input.txt defining input  
import_data_file model/predictions.txt defining predictions
```

```
thm input_def  
thm predictions_def
```

```
value ⟨(checkget_result_singleton 0.15 (predictdigraph_single compass.NeuralNet (map_of_list (input!o)) E12_16))  
(Some (predictions!o!o)))⟩  
value ⟨(checkget_result_singleton 0.15 (predictdigraph_single compass.NeuralNet (map_of_list (input!o)) E12_16))  
(Some (predictions!1!o)))⟩  
value ⟨(checkget_result_singleton 0.15 (predictdigraph_single compass.NeuralNet (map_of_list (input!o)) E12_16))  
(Some (predictions!2!o)))⟩  
value ⟨(checkget_result_singleton 0.15 (predictdigraph_single compass.NeuralNet (map_of_list (input!o)) E12_16))  
(Some (predictions!3!o)))⟩
```

lemma compass_truth_table_predict:

```
⟨(predictdigraph_single compass.NeuralNet (map_of_list (input!o)) E12_16) ≈[0.0001] (Some (predictions!o!o))⟩
```

```

<(predictdigraph_single compass.NeuralNet (map_of_list (input!1)) E12_16) ≈ [0.0001] ≈s (Some (predictions!1!o))>
<(predictdigraph_single compass.NeuralNet (map_of_list (input!2)) E12_16) ≈ [0.0001] ≈s (Some (predictions!2!o))>
<(predictdigraph_single compass.NeuralNet (map_of_list (input!3)) E12_16) ≈ [0.0001] ≈s (Some (predictions!3!o))>
by(normalization)+

```

lemma *compass_truth_table_predict'*:

```

<(predictdigraph_single_list compass.NeuralNet (input!o) ≈ [0.0001] ≈l (Some (predictions!o)))>
<(predictdigraph_single_list compass.NeuralNet (input!1) ≈ [0.0001] ≈l (Some (predictions!1)))>
<(predictdigraph_single_list compass.NeuralNet (input!2) ≈ [0.0001] ≈l (Some (predictions!2)))>
<(predictdigraph_single_list compass.NeuralNet (input!3) ≈ [0.0001] ≈l (Some (predictions!3)))>
by(normalization)+

```

Multi Encoding **declare** $[[nn_proof_mode = eval]]$

import TensorFlow *compass_multi* **file** model/model.json **as** digraph

declare $[[nn_proof_mode = nbe]]$

thm *compass_multi.neuron_defs*

thm *compass_multi.Neurons_def*

thm *compass_multi.edge_defs*

thm *compass_multi.Edges_def*

thm *compass_multi.Graph_def*

thm *compass_multi.verts_set_conv*

thm *compass_multi.edges_set_conv*

thm *compass_multi.φ_compass_multi.simps*

thm *compass_multi.NeuralNet_def*

thm *compass_multi.nn_pregraph_axioms*

thm *compass_multi.neural_network_digraph_axioms*

Checking Equivalence of Manual Definitions and Automated Import **lemma** *Neurons_equiv*: *compass.Neurons* = *m_Neurons*

by(normalization)

lemma *Edges_equiv*: *compass.Edges* = *m_Edges*

by(normalization)

lemma *Graph_equiv*: *compass.Graph* = *m_Graph*

unfolding *compass.Graph_def* *m_Graph_def*

by (simp add: *Edges_equiv*)

lemma *φ_equiv*: *compass.φ_compass* *f* = *m_φ_compass* *f*

by(cases *f*, *simp_all*)

lemma *NeuralNet_equiv*: *compass.NeuralNet* = *m_NeuralNet*

unfolding *compass.NeuralNet_def* *m_NeuralNet_def*

by(auto simp add: *Graph_equiv* *φ_equiv*)

lemma \langle *predict_{digraph_single_list} compass.NeuralNet* = *predict_{digraph_single_list} m_NeuralNet* \rangle

using *NeuralNet_equiv* **by** simp

```

Code Evaluation definition NW = [0::nat ↦ 1, 1::nat ↦ 1, 2::nat ↦ 1,
  3::nat ↦ 1, 4::nat ↦ 1, 5::nat ↦ 0,
  6::nat ↦ 1, 7::nat ↦ 0, 8::nat ↦ 1]

```

```

definition <eval_compass = predictdigraph_single compass.NeuralNet NW compass.Edges.E12_16>

```

```

end

```

8.1.2 Neural Networks as List of Layers using List Types (⊞ Compass_Layers_List)

```

theory

```

```

  Compass_Layers_List

```

```

imports

```

```

  Neural_Networks.NN_Layers_List_Main

```

```

begin

```

Manual Definition

```

Definition: Activation Tab fun m_φ_compass :: <activationmulti ⇒ (real list ⇒ real list) option> where
  <m_φ_compass mldentity = Some (map identity)>
  | <m_φ_compass _ = None>

```

```

Definition: Layers definition m_dense_input = In (|name = STR "dense_input", units = 9|)

```

```

definition m_dense =

```

```

  Dense

```

```

  (|name = STR "dense", units = 3, φ = mldentity,
    β = [9153944253921509 / 10000000000000000, - 959978699684143 / 10000000000000000, 7840137928724289 /
    100000000000000000],
    ω = [[- 28655481338500977 / 10000000000000000, 1398887038230896 / 10000000000000000, - 4396021068096161 /
    100000000000000000,
      - 3206970691680908 / 100000000000000000, - 25562143325805664 / 100000000000000000, 11852015256881714 /
    100000000000000000,
      6039865016937256 / 100000000000000000, - 16825008392333984 / 100000000000000000, - 413370318710804 /
    100000000000000000],
    [24456006288528442 / 100000000000000000, - 11522198915481567 / 100000000000000000, 4993317425251007 /
    100000000000000000,
      - 17345187664031982 / 100000000000000000, 48335906863212585 / 100000000000000000, 1511125922203064 /
    100000000000000000,
      - 36204618215560913 / 100000000000000000, 9508050084114075 / 100000000000000000, - 3617756962776184 /
    100000000000000000],
    [704086497426033 / 100000000000000000, - 51195383071899414 / 100000000000000000, - 34204763174057007 /
    100000000000000000,
      - 72454833984375 / 100000000000000000, - 33541640639305115 / 100000000000000000, 12738076448440552 /
    100000000000000000,
      7601173520088196 / 100000000000000000, - 2638514041900635 / 100000000000000000, - 5478811264038086 /
    100000000000000000]]])

```

```

definition m_dense_2 =

```

```

  Dense

```

```

  (|name = STR "dense_2", units = 4, φ = mldentity,
    β = [39810407906770706 / 100000000000000000, 874686986207962 / 100000000000000000, - 4944610595703125 /
    100000000000000000,
      - 5116363242268562 / 100000000000000000],
    ω = [[[ (9063153862953186 / 100000000000000000::real), - 142851984500885 / 100000000000000000, -
    10823805332183838 / 100000000000000000],

```

```

[17654908895492554 / 1000000000000000, 1934271901845932 / 1000000000000000, 1214023232460022 /
1000000000000000],
[- 17099318504333496 / 1000000000000000, - 7595149427652359 / 1000000000000000, - 12841564416885376
/ 1000000000000000],
[- 615866482257843 / 1000000000000000, 1532884955406189 / 1000000000000000, 17860114574432373 /
1000000000000000]]))

```

```

definition m_OUTPUT ≡ Out (⟦name = STR "OUTPUT", units = 4⟧)

```

lemmas

```

m_layer_defs = m_dense_input_def m_dense_def m_dense_2_def m_OUTPUT_def

```

definition

```

⟦m_Layers = [m_dense_input, m_dense, m_dense_2, m_OUTPUT]⟧

```

Definition: Neural Network definition

```

⟦m_NeuralNet = (⟦layers = m_Layers, activation_tab = m_φ_compass⟧)⟧

```

Locale Interpretations lemma m_φ_{ran} : $\langle \text{ran } m_\varphi_{\text{compass}} = \{\text{map identity}\} \rangle$

```

apply(auto simp add: ran_def ranI)[1]
apply(elim m_φ_compass.elims)
apply((auto simp add: ran_def ranI)[1])+
apply(meson bot.extremum insert_subsetI ranI m_φ_compass.simps)+
done

```

interpretation nn_{nor} : $\text{neural_network_sequential_layers}_l$ $m_NeuralNet$

```

apply(simp add: neural_network_sequential_layers_l_def)
apply(safe)
subgoal by(simp add: m_layer_defs m_NeuralNet_def m_Layers_def)
subgoal by(simp add: m_layer_defs m_NeuralNet_def m_Layers_def)
subgoal by(simp add: m_layer_defs m_NeuralNet_def m_Layers_def)
subgoal by(simp add: m_φ_ran valid_activation_tab_l_def m_NeuralNet_def)
subgoal by(normalization)
done

```

TensorFlow Import

```

declare[[nn_proof_mode = eval]]
import TensorFlow compass file model/model.json as seq_layer_list
declare[[nn_proof_mode = nbe]]

```

```

thm compass.Layers.dense_input_def
thm compass.Layers.dense_def
thm compass.Layers.OUTPUT_def
thm compass.layer_defs
thm compass.Layers_def
thm compass.φ_compass.simps
thm compass.NeuralNet_def

```

```

thm compass.neural_network_sequential_layers_l_axioms

```

```

import_data_file model/input.txt defining input
import_data_file model/predictions.txt defining predictions

```

```
thm input_def
thm predictions_def
```

```
lemmas digits_defs = compass.Layers_def
lemmas activation_defs = identity_def
```

```
value predict_seq_layer_l NeuralNet (input!0)
```

```
value <checkget_result_list 0.001 (predict_seq_layer_l NeuralNet (input!0)) (Some (predictions!0))>
value <checkget_result_list 0.001 (predict_seq_layer_l NeuralNet (input!1)) (Some (predictions!1))>
value <checkget_result_list 0.001 (predict_seq_layer_l NeuralNet (input!2)) (Some (predictions!2))>
value <checkget_result_list 0.001 (predict_seq_layer_l NeuralNet (input!3)) (Some (predictions!3))>
```

We convince ourselves that our Isabelle representation complies with the TensorFlow network by generating the same prediction, within 0.001 (accounted for as Isabelle uses perfect mathematical reals whereas TensorFlow uses 32-bit floating point numbers)

```
lemma compass_predictions:
```

```
<(predict_seq_layer_l NeuralNet (input!0)) ≈[0.001]≈l (Some (predictions!0))>
<(predict_seq_layer_l NeuralNet (input!1)) ≈[0.001]≈l (Some (predictions!1))>
<(predict_seq_layer_l NeuralNet (input!2)) ≈[0.001]≈l (Some (predictions!2))>
<(predict_seq_layer_l NeuralNet (input!3)) ≈[0.001]≈l (Some (predictions!3))>
by(normalization)+
```

```
lemma <0.000001 |=l {input} (predict_seq_layer_l NeuralNet) {predictions}> by eval
```

```
lemma activation[simp]: <activation_tab NeuralNet = compass.φ_compass >
by(simp add: compass.Layers_def compass.layer_defs compass.NeuralNet_def)
```

```
lemma layers[simp]: <layers NeuralNet = [dense_input, Layers.dense, dense_2, OUTPUT] >
by(simp add: compass.Layers_def compass.NeuralNet_def)
```

```
lemma input[simp]: <in_deg_NN NeuralNet = 9 >
by(simp add: compass.Layers_def compass.NeuralNet_def in_deg_NN_def dense_input_def)
```

```
import_data_file model/compass.txt defining compass
```

```
definition classify_as :: <real list ⇒ nat ⇒ bool> where
<classify_as xs n = (Option.bind (predict_seq_layer_l compass.NeuralNet xs) Prediction_Utils.pos_of_max = Some n)>
```

```
lemma co[simp]: compass!0 = [1,1,1,
    1,1,0,
    1,0,1]
by (simp add: compass_def)
```

```
lemma c1[simp]: compass!1 = [1,1,1,
    0,1,1,
    1,0,1]
by (simp add: compass_def)
```

```
lemma c2[simp]: compass!2 = [1,0,1,
    0,1,1,
    1,1,1]
by (simp add: compass_def)
```

```

lemma c3[simp]: compass!3 = [1,0,1,
    1,1,0,
    1,1,1]
by (simp add: compass_def)

lemma classify_NW: <classify_as (compass!0) 0>
by eval
lemma classify_NE: <classify_as (compass!1) 1>
by eval

lemma classify_SE: <classify_as (compass!2) 2>
by eval
lemma classify_SW: <classify_as (compass!3) 3>
by eval

lemma compass_img_defined: <((predict_seq_layer_l compass.NeuralNet xs) ≠ None) = (length xs = 9)>
using compass.neural_network_sequential_layers_l_axioms
neural_network_sequential_layers_l_img_None[of compass.NeuralNet]
by simp

end

```

8.1.3 Neural Networks as List of Layers using Matrix Types (📄 Compass_Layers_Matrix)

theory

Compass_Layers_Matrix

imports

Neural_Networks.NN_Layers_Matrix_Main

Jordan_Normal_Form.Matrix_Impl

Prediction_Utils_Matrix

begin

Infrastructure

definition

<checkget_result_matrix ε prediction input expectations = checkget_result_list ε (map_option list_of_vec (prediction (Some (vec_of_list (input)))))) (Some (expectations))>

definition predict_def[simp]: <predict N x = (map_option list_of_vec (predict_seq_layer_m N (vec_of_list x)))>

Manual Definition

Definition: Activation Tab fun m_φ_compass where

<m_φ_compass mIdentity = Some (map_vec identity)>

| <m_φ_compass _ = None>

Definition: Layers definition m_dense_input = In (name = STR "dense_input", units = 9)

definition m_dense =

Dense

(name = STR "dense", units = 3, φ = mIdentity,

β = vec_of_list [9153944253921509 / 10000000000000000, - 959978699684143 / 10000000000000000, 7840137928724289 / 10000000000000000],

ω = mat_of_cols_list 9

```

[[[- 28655481338500977 / 10000000000000000, 1398887038230896 / 10000000000000000, - 4396021068096161
/ 10000000000000000,
- 3206970691680908 / 10000000000000000, - 25562143325805664 / 10000000000000000, 11852015256881714
/ 10000000000000000,
6039865016937256 / 10000000000000000, - 16825008392333984 / 10000000000000000, - 413370318710804 /
10000000000000000],
[24456006288528442 / 10000000000000000, - 11522198915481567 / 10000000000000000, 4993317425251007 /
10000000000000000,
- 17345187664031982 / 10000000000000000, 48335906863212585 / 10000000000000000, 1511125922203064 /
10000000000000000,
- 36204618215560913 / 10000000000000000, 9508050084114075 / 10000000000000000, - 3617756962776184
/ 10000000000000000],
[704086497426033 / 10000000000000000, - 51195383071899414 / 10000000000000000, - 34204763174057007
/ 10000000000000000,
- 72454833984375 / 10000000000000000, - 33541640639305115 / 10000000000000000, 12738076448440552 /
10000000000000000,
7601173520088196 / 10000000000000000, - 2638514041900635 / 10000000000000000, - 5478811264038086 /
10000000000000000]]])

```

definition $m_dense_2 =$

```

Dense
(|name = STR "dense_2", units = 4,  $\varphi = m\_identity$ ,
 $\beta = \text{vec\_of\_list}$  [39810407906770706 / 10000000000000000, 874686986207962 / 10000000000000000, -
4944610595703125 / 10000000000000000,
- 5116363242268562 / 10000000000000000],
 $\omega = \text{mat\_of\_cols\_list}$  3
[[[(9063153862953186 / 10000000000000000::real), - 142851984500885 / 10000000000000000, - 10823805332183838
/ 10000000000000000],
[17654908895492554 / 10000000000000000, 1934271901845932 / 10000000000000000, 1214023232460022 /
10000000000000000],
[- 17099318504333496 / 10000000000000000, - 7595149427652359 / 10000000000000000, - 12841564416885376
/ 10000000000000000],
[- 615866482257843 / 10000000000000000, 1532884955406189 / 10000000000000000, 17860114574432373 /
10000000000000000]]])

```

definition $m_OUTPUT \equiv \text{Out}$ ($|name = \text{STR "OUTPUT"$, $units = 4$)

lemmas

$m_layer_defs = m_dense_input_def\ m_dense_def\ m_dense_2_def\ m_OUTPUT_def$

definition

$\langle m_Layers = [m_dense_input, m_dense, m_dense_2, m_OUTPUT] \rangle$

Definition: Neural Network **definition**

$\langle m_NeuralNet = (|layers = m_Layers, activation_tab = m_varphi_compass) \rangle$

Locale Interpretations **lemma** m_varphi_ran : $\langle ran\ m_varphi_compass = \{map_vec\ identity\} \rangle$

```

apply(auto simp add:  $ran\_def\ ranI$ )[1]
apply(elim  $m\_varphi\_compass.elims$ )
apply((auto simp add:  $ran\_def\ ranI$ )[1])+
apply(meson bot.extremum insert_subsetI  $ranI\ m\_varphi\_compass.simps$ )+
done

```

interpretation $nn_{n.or}$: $neural_network_sequential_layers_m\ m_NeuralNet$

apply(**simp add**: $neural_network_sequential_layers_m_def$)

```

apply(safe)
subgoal by(simp add: m_layer_defs m_NeuralNet_def m_Layers_def)
subgoal by(simp add: m_layer_defs m_NeuralNet_def m_Layers_def)
subgoal by(simp add: m_layer_defs m_NeuralNet_def m_Layers_def)
subgoal by(simp add: m_φ_ran valid_activation_tab_m_def m_NeuralNet_def)
subgoal by(normalization)
done

```

TensorFlow Import

```

declare[[nn_proof_mode = eval]]
import_TensorFlow compass file model/model.json as seq_layer_matrix
declare[[nn_proof_mode = nbe]]

```

```

find_theorems name:compass. name:φ name:ran

```

```

thm compass.Layers.dense_input_def
thm compass.Layers.dense_def
thm compass.Layers.OUTPUT_def
thm compass.layer_defs
thm compass.Layers_def
thm compass.φ_compass.simps
thm compass.NeuralNet_def
thm compass.neural_network_sequential_layers_m_axioms

```

```

import_data_file model/input.txt defining input
import_data_file model/predictions.txt defining predictions

```

```

thm input_def
thm predictions_def

```

```

value <(predictseq_layer_m compass.NeuralNet) (vec_of_list(input!o))>

```

```

value <checkget_result_matrix 0.001 (predictseq_layer_m NeuralNet o the) (input!o) (predictions!o)>

```

```

value <checkget_result_matrix 0.001 (predictseq_layer_m NeuralNet o the) (input!1) (predictions!1)>

```

```

value <checkget_result_matrix 0.001 (predictseq_layer_m NeuralNet o the) (input!2) (predictions!2)>

```

```

value <checkget_result_matrix 0.001 (predictseq_layer_m NeuralNet o the) (input!3) (predictions!3)>

```

We convince ourselves that our Isabelle representation complies with the TensorFlow network by generating the same prediction, within 0.001 (accounted for as Isabelle uses perfect mathematical reals whereas TensorFlow uses 32-bit floating point numbers)

```

lemma compass_predictions:

```

```

  <(map_option list_of_vec (predictseq_layer_m NeuralNet ((vec_of_list (input!o))))> ≈[0.001]≈l (Some (predictions!o))>

```

```

  <(map_option list_of_vec (predictseq_layer_m NeuralNet ((vec_of_list (input!1))))> ≈[0.001]≈l (Some (predictions!1))>

```

```

  <(map_option list_of_vec (predictseq_layer_m NeuralNet ((vec_of_list (input!2))))> ≈[0.001]≈l (Some (predictions!2))>

```

```

  <(map_option list_of_vec (predictseq_layer_m NeuralNet ((vec_of_list (input!3))))> ≈[0.001]≈l (Some (predictions!3))>

```

```

by eval+

```

```

lemma <0.000001 ⊨l {input} (predict NeuralNet) {predictions}>

```

by eval

lemma *activation*[simp]: $\langle \text{activation_tab } \text{NeuralNet} = \text{compass}.\varphi_{\text{compass}} \rangle$
by(simp add: *compass.Layers_def compass.layer_defs compass.NeuralNet_def*)

lemma *layers*[simp]: $\langle \text{layers } \text{NeuralNet} = [\text{dense_input}, \text{Layers.dense}, \text{dense_2}, \text{OUTPUT}] \rangle$
by(simp add: *compass.Layers_def compass.NeuralNet_def*)

lemma *input*[simp]: $\langle \text{in_deg_NN } \text{NeuralNet} = 9 \rangle$
by(simp add: *compass.Layers_def compass.NeuralNet_def in_deg_NN_def dense_input_def*)

import_data_file *model/compass.txt* **defining** *compass*

lemma *co*[simp]: $\text{compass!o} = [1,1,1,$
 $1,1,0,$
 $1,0,1]$
by (simp add: *compass_def*)

lemma *c1*[simp]: $\text{compass!1} = [1,1,1,$
 $0,1,1,$
 $1,0,1]$
by (simp add: *compass_def*)

lemma *c2*[simp]: $\text{compass!2} = [1,0,1,$
 $0,1,1,$
 $1,1,1]$
by (simp add: *compass_def*)

lemma *c3*[simp]: $\text{compass!3} = [1,0,1,$
 $1,1,0,$
 $1,1,1]$
by (simp add: *compass_def*)

lemma *compass_img_defined*: $\langle ((\text{predict}_{\text{seq_layer_m}} \text{compass.NeuralNet } \text{xs}) \neq \text{None}) = (\text{length } (\text{list_of_vec } \text{xs}) = 9) \rangle$
using *compass.neural_network_sequential_layers_m_axioms*
 neural_network_sequential_layers_m_img_None[of *compass.NeuralNet*]
by *simp*

definition *classify_as* :: $\langle \text{real Matrix.vec} \Rightarrow \text{nat} \Rightarrow \text{bool} \rangle$ **where**
 $\langle \text{classify_as } \text{xs } n = (\text{Option.bind } (\text{predict}_{\text{seq_layer_m}} \text{compass.NeuralNet } \text{xs}) \text{ pos_of_max} = \text{Some } n) \rangle$

lemma *classify_NW*: $\langle \text{classify_as } (\text{vec_of_list}(\text{compass!o})) \text{ o} \rangle$
by *eval*

lemma *classify_NE*: $\langle \text{classify_as } (\text{vec_of_list}(\text{compass!1})) \text{ 1} \rangle$
by *eval*

lemma *classify_SE*: $\langle \text{classify_as } (\text{vec_of_list}(\text{compass!2})) \text{ 2} \rangle$
by *eval*

lemma *classify_SW*: $\langle \text{classify_as } (\text{vec_of_list}(\text{compass!3})) \text{ 3} \rangle$
by *eval*

end

8.2 Line Classification Model (Grid_Layers) (Grid_Layers)

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., Figure 8.1a), vertical line (e.g., Figure 8.1b), or no line (Figure 8.1c).



Figure 8.1: Example input images to our classification problem.

Traditionally, textbooks (e.g., [2]) 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 the respective weights w_{k_0} to $w_{k_{m-1}}$ the neuron computes the output

$$y_k = \varphi \left(\beta \sum_{j=0}^m w_{k_j} x_{k_j} \right) \quad (8.1)$$

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].

Figure 8.2 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 has three outputs, one for each possible class (horizontal line, vertical line, 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. This is 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 neural network that reliably classifies black lines in a given 2×2 image with 100% accuracy. While this sounds great,

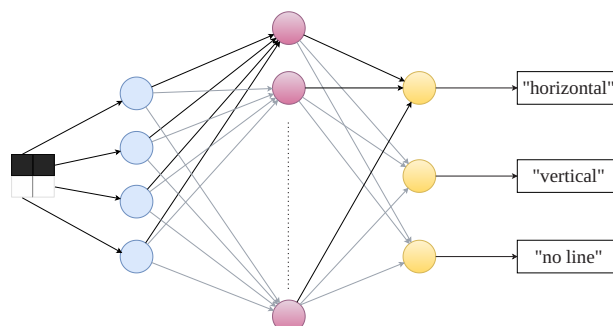


Figure 8.2: Neural network for classifying lines in 2×2 pixel grey scale images.

the neural network is not very resilient to changes to its input values. Consider, for example, Figure 8.1d: a human expert would, very likely, classify this image as “no line”. Yet our neural network classifies this as a horizontal line, even though the right upper pixel is only light grey with a numerical value of 0.05, much closer to white than to black. Such a misclassification is usually called an *adversarial example*. If such a network is used in a safety or security critical applications, e.g., for classifying street signs, such misclassifications can be life-threatening.

```
theory
  Grid_Layers
imports
  NN_Layers_List_Main
begin
end
```

8.2.1 Layer-based Modelling using List Types (Grid_Layers_List)

```
theory
  Grid_Layers_List
imports
  NN_Layers_List_Main
  Grid_Layers
begin

declare[[nn_proof_mode = eval]]
import_TensorFlow grid file model/trained-model_binary-step_linear/model.json
  as seq_layer_list
declare[[nn_proof_mode = nbe]]
```

Our new Isabelle/Isar command `import_TensorFlow` encodes the neural network model stored in the file `model.json` as sequence of layers, i.e., the formal encoding we developed. Our datatype package also proves that the imported model complies with the requirements of our formal model as well as proves various auxiliary properties (e.g., conversion between different representations) that can be useful during interactive verification.

```
import_data_file model/trained-model_binary-step_linear/input_small.txt
  defining inputs_small
import_data_file model/trained-model_binary-step_linear/expectations_small.txt
  defining expectations_small

import_data_file model/trained-model_binary-step_linear/input.txt
  defining inputs
import_data_file model/trained-model_binary-step_linear/expectations.txt
  defining expectations
import_data_file model/trained-model_binary-step_linear/predictions.txt
  defining predictions
```

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.

We can import text files containing NumPy arrays of our test inputs, expectations and predictions from our trained TensorFlow network.

```

thm grid.Layers_def
thm grid.Layers.dense_input_def
thm grid.Layers.OUTPUT_def
thm grid.layer_defs
thm grid.Layers_def
thm grid.φ_grid.simps
thm grid.NeuralNet_def
thm inputs_def
thm predictions_def

```

```

lemmas grid_defs = grid.Layers_def grid.layer_defs grid.NeuralNet_def
lemmas activation_defs = identity_def binary_step_def

```

lemma grid_closed [simp]:

```

⟨predictseq_layer_l grid.NeuralNet xs = (case xs of
[x3, x2, x1, x0] ⇒ let y = 2 * (x3 + (x2 * 2 + (x1 * 4 + x0 * 8)))
in Some (
  [(if y - 7 ≤ 0 then 0 else 1) +
   ((if y - 11 ≤ 0 then 0 else 1) + ((if y - 21 ≤ 0 then 0 else 1)
   + ((if y - 25 ≤ 0 then 0 else 1) - (if y - 23 ≤ 0 then 0 else 1))
   - (if y - 19 ≤ 0 then 0 else 1)) - (if y - 9 ≤ 0 then 0 else 1))
   - (if y - 5 ≤ 0 then 0 else 1) + 1,
  (if y - 5 ≤ 0 then 0 else 1) + ((if y - 23 ≤ 0 then 0 else 1)
  - (if y - 25 ≤ 0 then 0 else 1) - (if y - 7 ≤ 0 then 0 else 1)),
  (if y - 9 ≤ 0 then 0 else 1) + ((if y - 19 ≤ 0 then 0 else 1)
  - (if y - 21 ≤ 0 then 0 else 1) - (if y - 11 ≤ 0 then 0 else 1))]
)
| _ ⇒ None⟩
by(auto simp add:grid_defs predictseq_layer_l_def activation_defs Let_def split:list.split)

```

lemma grid_img_defined: ⟨((predict_{seq_layer_l} grid.NeuralNet xs) ≠ None) = (length xs = 4)⟩
by(auto simp add:Let_def split:list.split)

lemma grid_img_defined': ⟨(∃ y. (predict_{seq_layer_l} grid.NeuralNet xs) = Some y) = (length xs = 4)⟩
using grid_img_defined **by** auto

lemma grid_image:

```

assumes ⟨(predictseq_layer_l grid.NeuralNet xs) ≠ None⟩
shows ⟨the (predictseq_layer_l grid.NeuralNet xs) ∈ { [0, 0, 1],
  [0, 1, 0],
  [1, 0, 0] }⟩

```

```

using assms grid_img_defined apply simp
by(auto simp add:Let_def split:list.split)

```

lemma grid_image_approx:

```

⟨ran (predictseq_layer_l grid.NeuralNet) ⊆ { [0, 0, 1], [0, 1, 0], [1, 0, 0] }⟩
apply(simp only:ran_def)
using grid_image by force

```

The lemma `grid_image_approx` shows that the output of the classification is never ambiguous (i.e., two or more classification output having the value 1).

lemma grid_dom: ⟨dom (predict_{seq_layer_l} grid.NeuralNet) = {a. length a = 4}⟩
by (simp add:grid_defs predict_{seq_layer_l}_def activation_defs dom_def)

```
definition range_of x = (if x = (0::real) then {0..0.04::real} else {0.96..1})
```

```
lemma <x3 ∈ {0.96..1.00} ∧ x2 ∈ {0.96..1.00}
  ∧ x1 ∈ {0.00..0.04} ∧ x0 ∈ {0.00..0.04} ⇒ predict_seq_layer_l grid.NeuralNet [x3, x2, x1, x0] = Some [0, 1, 0]>
by(simp add: Let_def)
```

```
lemma <x3 ∈ {0.00..0.04} ∧ x2 ∈ {0.00..0.04}
  ∧ x1 ∈ {0.96..1.00} ∧ x0 ∈ {0.96..1.00} ⇒ predict_seq_layer_l grid.NeuralNet [x3, x2, x1, x0] = Some [0, 1, 0]>
by(simp add: Let_def)
```

```
lemma <x3 ∈ {0.95..1.00} ∧ x2 ∈ {0.00..0.05}
  ∧ x1 ∈ {0.95..1.00} ∧ x0 ∈ {0.00..0.05} ⇒ predict_seq_layer_l grid.NeuralNet [x3, x2, x1, x0] = Some [0, 0, 1]>
by(simp add: Let_def)
```

```
lemma <x3 ∈ {0.00..0.1} ∧ x2 ∈ {0.96..1.00}
  ∧ x1 ∈ {0.00..0.1} ∧ x0 ∈ {0.96..1.00} ⇒ predict_seq_layer_l grid.NeuralNet [x3, x2, x1, x0] = Some [0, 0, 1]>
by(simp add: Let_def)
```

A common definition of safety in neural networks is the requirement that small changes to an input should not change the classification. For this grid example, we express such a verification goal as shown above, where we set a small bound of noise on the input, and verify that the output classification remains constant.

```
lemma grid_meets_predictions:
  <|=il {inputs} (predict_seq_layer_l grid.NeuralNet) {intervals_of_l 0.000001 predictions}>
by (simp add: ensure_testdata_interval_list_def upper_Interval lower_Interval predictions_def
  intervals_of_l_def inputs_def in_set_interval)
```

```
lemma grid_meets_expectations_max_classifier:
  <|=l {inputs_small} (predict_seq_layer_l grid.NeuralNet) {expectations_small}>
by(simp add: ensure_testdata_max_list_def expectations_small_def inputs_small_def)
```

```
lemma grid_min_delta_classifier:
  <1.0 |= predict_seq_layer_l grid.NeuralNet>
unfolding ensure_delta_min_def Prediction_Utils.δ_min_def max_list_def
using grid_image_approx
by auto
```

The lemmas *grid_meets_predictions*, *grid_meets_expectations_max_classifier* and *grid_min_delta_classifier* show that our definition of the grid neural network computes the same prediction as the TensorFlow trained network.

```
end
```

8.2.2 Layer-based Modelling using List Types (Grid_Layers_Matrix)

```
theory
  Grid_Layers_Matrix
imports
  Grid_Layers
  NN_Layers_Matrix_Main
  Jordan_Normal_Form.Matrix_Impl
begin

declare[[nn_proof_mode = eval]]
import TensorFlow grid file model/trained-model_binary-step_linear/model.json
  as seq_layer_matrix
declare[[nn_proof_mode = nbe]]
```

Our new Isabelle/Isar command `import_TensorFlow` encodes the neural network model stored in the file `model.json` as sequence of layers, i.e., the formal encoding we developed. Our datatype package also proves that the imported model complies with the requirements of our formal model as well as proves various auxiliary properties (e.g., conversion between different representations) that can be useful during interactive verification.

```
import_data_file model/trained-model_binary-step_linear/input_small.txt
defining inputs_small
import_data_file model/trained-model_binary-step_linear/expectations_small.txt
defining expectations_small
```

```
import_data_file model/trained-model_binary-step_linear/input.txt
defining inputs
import_data_file model/trained-model_binary-step_linear/expectations.txt
defining expectations
import_data_file model/trained-model_binary-step_linear/predictions.txt
defining predictions
```

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.

We can import text files containing NumPy arrays of our test inputs, expectations and predictions from our trained TensorFlow network.

```
thm grid.Layers_def
thm grid.Layers.dense_input_def
thm grid.Layers.OUTPUT_def
thm grid.layer_defs
thm grid.Layers_def
thm grid. $\varphi$ _grid.simps
thm grid.NeuralNet_def
thm inputs_def
thm predictions_def
```

```
lemmas grid_defs = grid.Layers_def grid.layer_defs grid.NeuralNet_def
lemmas activation_defs = identity_def binary_step_def
```

Proving using the matrix prediction function.

```
lemma grid_closed_mat [simp]:
   $\langle \text{predict}_{\text{seq\_layer\_m}} \text{grid.NeuralNet} (\text{vec\_of\_list } xs) = (\text{case } xs \text{ of}$ 
   $[x_3, x_2, x_1, x_0] \Rightarrow \text{let } y = 2 * (x_3 + (x_2 * 2 + (x_1 * 4 + x_0 * 8)))$ 
   $\text{in Some (}$ 
   $\text{vec\_of\_list}[(\text{if } y - 7 \leq 0 \text{ then } 0 \text{ else } 1) +$ 
   $((\text{if } y - 11 \leq 0 \text{ then } 0 \text{ else } 1) + ((\text{if } y - 21 \leq 0 \text{ then } 0 \text{ else } 1)$ 
   $+ ((\text{if } y - 25 \leq 0 \text{ then } 0 \text{ else } 1) - (\text{if } y - 23 \leq 0 \text{ then } 0 \text{ else } 1))$ 
   $- (\text{if } y - 19 \leq 0 \text{ then } 0 \text{ else } 1)) - (\text{if } y - 9 \leq 0 \text{ then } 0 \text{ else } 1))$ 
   $- (\text{if } y - 5 \leq 0 \text{ then } 0 \text{ else } 1) + 1,$ 
   $(\text{if } y - 5 \leq 0 \text{ then } 0 \text{ else } 1) + ((\text{if } y - 23 \leq 0 \text{ then } 0 \text{ else } 1)$ 
   $- (\text{if } y - 25 \leq 0 \text{ then } 0 \text{ else } 1) - (\text{if } y - 7 \leq 0 \text{ then } 0 \text{ else } 1)),$ 
   $(\text{if } y - 9 \leq 0 \text{ then } 0 \text{ else } 1) + ((\text{if } y - 19 \leq 0 \text{ then } 0 \text{ else } 1)$ 
   $- (\text{if } y - 21 \leq 0 \text{ then } 0 \text{ else } 1) - (\text{if } y - 11 \leq 0 \text{ then } 0 \text{ else } 1))]$ 
   $\rangle$ 
  )
```

```

|_  $\Rightarrow$  None)›
apply (auto simp add: predictseq_layer_m_def grid_defs activation_defs split:list.split)[1]
apply(normalization)
  by (simp_all add: grid_defs predictseq_layer_m_def activation_defs dom_def dim_col_mat_of_col_list
dim_row_mat_of_col_list)

```

```

lemma grid_img_defined_mat:  $\langle ((\text{predict}_{\text{seq\_layer\_m}} \text{grid.NeuralNet } (\text{vec\_of\_list } xs)) \neq \text{None}) = (\text{dim\_vec } (\text{vec\_of\_list } xs) = 4) \rangle$ 
by(auto simp add:Let_def split:list.split)

```

```

lemma grid_img_defined_mat':  $\langle (\exists y. (\text{predict}_{\text{seq\_layer\_m}} \text{grid.NeuralNet } (\text{vec\_of\_list } xs)) = \text{Some } (\text{vec\_of\_list } y)) = (\text{dim\_vec } (\text{vec\_of\_list } xs) = 4) \rangle$ 
proof(cases (dim_vec (vec_of_list xs)) = 4)
  case True
  then show ?thesis
    by (metis grid_img_defined_mat option.exhaust vec_list)
next
  case False
  then show ?thesis
    using grid_img_defined_mat by blast
qed

```

```

lemma grid_image_mat:
assumes  $\langle \text{predict}_{\text{seq\_layer\_m}} \text{grid.NeuralNet } (\text{vec\_of\_list } xs) = \text{Some } y \rangle$ 
shows  $\langle y \in \{ \text{vec\_of\_list } [0, 0, 1], \text{vec\_of\_list } [0, 1, 0], \text{vec\_of\_list } [1, 0, 0] \} \rangle$ 
using assms grid_img_defined_mat
apply(simp split:list.splits)
unfolding Let_def vec_of_list_def vCons_def
apply(simp)
by auto

```

```

lemma ran_aux:
assumes  $\langle \forall x. f x \neq \text{None} \longrightarrow \text{the } (f x) \in Y \rangle$ 
shows  $\langle \text{ran } (f) \subseteq Y \rangle$ 
unfolding ran_def using assms
by force

```

```

lemma grid_image_approx_mat:
 $\langle \text{ran } (\lambda x. \text{predict}_{\text{seq\_layer\_m}} \text{grid.NeuralNet } (\text{vec\_of\_list } x)) \subseteq \{ \text{vec\_of\_list } [0, 0, 1], \text{vec\_of\_list } [0, 1, 0], \text{vec\_of\_list } [1, 0, 0] \} \rangle$ 
using grid_image_mat ran_aux
by (metis (no_types, lifting) option.exhaust_sel)

```

The lemma `grid_image_approx` shows that the output of the classification is never ambiguous (i.e., two or more classification output having the value 1).

```

lemma grid_dom_mat:  $\langle \text{dom } (\lambda x. \text{predict}_{\text{seq\_layer\_m}} \text{grid.NeuralNet } (\text{vec\_of\_list } x)) = \{ a. \text{length } a = 4 \} \rangle$ 
by (simp add: grid_defs predictseq_layer_m_def activation_defs dom_def dim_col_mat_of_col_list
dim_row_mat_of_col_list)

```

```

definition range_of x = (if x = (0::real) then {0..0.04::real} else {0.96..1})

```

```

lemma  $\langle x_3 \in \{0.96..1.00\} \wedge x_2 \in \{0.96..1.00\} \wedge x_1 \in \{0.00..0.04\} \wedge x_0 \in \{0.00..0.04\} \implies \text{predict}_{\text{seq\_layer\_m}} \text{grid.NeuralNet } (\text{vec\_of\_list } [x_3, x_2, x_1, x_0]) = \text{Some}(\text{vec\_of\_list } [0, 1, 0]) \rangle$ 

```

```

by(simp add: Let_def, normalization, argo)
lemma  $\langle x_3 \in \{0.00..0.04\} \wedge x_2 \in \{0.00..0.04\}$ 
   $\wedge x_1 \in \{0.96..1.00\} \wedge x_0 \in \{0.96..1.00\} \implies \text{predict}_{\text{seq\_layer\_m}} \text{grid.NeuralNet} (\text{vec\_of\_list } [x_3, x_2, x_1, x_0]) =$ 
   $\text{Some}(\text{vec\_of\_list } [0, 1, 0]) \rangle$ 
by(simp add: Let_def, normalization, argo)
lemma  $\langle x_3 \in \{0.95..1.00\} \wedge x_2 \in \{0.00..0.05\}$ 
   $\wedge x_1 \in \{0.95..1.00\} \wedge x_0 \in \{0.00..0.05\} \implies \text{predict}_{\text{seq\_layer\_m}} \text{grid.NeuralNet} (\text{vec\_of\_list } [x_3, x_2, x_1, x_0]) =$ 
   $\text{Some}(\text{vec\_of\_list } [0, 0, 1]) \rangle$ 
by(simp add: Let_def, normalization, argo)
lemma  $\langle x_3 \in \{0.00..0.1\} \wedge x_2 \in \{0.96..1.00\}$ 
   $\wedge x_1 \in \{0.00..0.1\} \wedge x_0 \in \{0.96..1.00\} \implies \text{predict}_{\text{seq\_layer\_m}} \text{grid.NeuralNet} (\text{vec\_of\_list } [x_3, x_2, x_1, x_0]) =$ 
   $\text{Some}(\text{vec\_of\_list } [0, 0, 1]) \rangle$ 
by(simp add: Let_def, normalization, argo)

```

A common definition of safety in neural networks is the requirement that small changes to an input should not change the classification. For this grid example, we express such a verification goal as shown above, where we set a small bound of noise on the input, and verify that the output classification remains constant.

Proving using the list to matrix prediction

The following proofs on the grid example use our wrapper function that converts lists to vectors, uses the matrix based prediction function and converts the output back into a list

```

lemma grid_closed' [simp]:
   $\langle \text{predict}_{\text{seq\_layer\_m}}' \text{grid.NeuralNet } xs = (\text{case } xs \text{ of}$ 
     $[x_3, x_2, x_1, x_0] \Rightarrow \text{let } y = 2 * (x_3 + (x_2 * 2 + (x_1 * 4 + x_0 * 8)))$ 
     $\text{in } \text{Some} ($ 
       $[(\text{if } y - 7 \leq 0 \text{ then } 0 \text{ else } 1) +$ 
         $((\text{if } y - 11 \leq 0 \text{ then } 0 \text{ else } 1) + ((\text{if } y - 21 \leq 0 \text{ then } 0 \text{ else } 1)$ 
           $+ ((\text{if } y - 25 \leq 0 \text{ then } 0 \text{ else } 1) - (\text{if } y - 23 \leq 0 \text{ then } 0 \text{ else } 1))$ 
           $- (\text{if } y - 19 \leq 0 \text{ then } 0 \text{ else } 1)) - (\text{if } y - 9 \leq 0 \text{ then } 0 \text{ else } 1))$ 
           $- (\text{if } y - 5 \leq 0 \text{ then } 0 \text{ else } 1) + 1,$ 
         $(\text{if } y - 5 \leq 0 \text{ then } 0 \text{ else } 1) + ((\text{if } y - 23 \leq 0 \text{ then } 0 \text{ else } 1)$ 
           $- (\text{if } y - 25 \leq 0 \text{ then } 0 \text{ else } 1) - (\text{if } y - 7 \leq 0 \text{ then } 0 \text{ else } 1)),$ 
         $(\text{if } y - 9 \leq 0 \text{ then } 0 \text{ else } 1) + ((\text{if } y - 19 \leq 0 \text{ then } 0 \text{ else } 1)$ 
           $- (\text{if } y - 21 \leq 0 \text{ then } 0 \text{ else } 1) - (\text{if } y - 11 \leq 0 \text{ then } 0 \text{ else } 1))]$ 
       $)$ 
       $| \_ \Rightarrow \text{None} \rangle$ 
   $\rangle$ 
  apply (auto simp add: predict_seq_layer_m'_def predict_seq_layer_m_def grid_defs activation_defs split:list.split)[1]
  apply (normalization)
  apply (simp add: dim_col_mat_of_col_list dim_row_mat_of_col_list) +
done

```

```

lemma grid_img_defined':  $\langle ((\text{predict}_{\text{seq\_layer\_m}}' \text{grid.NeuralNet } xs) \neq \text{None}) = (\text{length } xs = 4) \rangle$ 
by(auto simp add: Let_def split:list.split)

```

```

lemma grid_img_defined:  $\langle (\exists y. (\text{predict}_{\text{seq\_layer\_m}}' \text{grid.NeuralNet } xs) = \text{Some } y) = (\text{length } xs = 4) \rangle$ 
using grid_img_defined' by auto

```

```

lemma grid_image:
  assumes  $\langle (\text{predict}_{\text{seq\_layer\_m}}' \text{grid.NeuralNet } xs) \neq \text{None} \rangle$ 
  shows  $\langle \text{the } (\text{predict}_{\text{seq\_layer\_m}}' \text{grid.NeuralNet } xs) \in \{ [0, 0, 1],$ 
     $[0, 1, 0],$ 
     $[1, 0, 0] \} \rangle$ 

```

```
using assms grid_img_defined' apply simp
by(auto simp add:Let_def split:list.split)
```

```
lemma grid_image_approx:
  <ran (predict_seq_layer_m' grid.NeuralNet) ⊆ {[0, 0, 1], [0, 1, 0], [1, 0, 0]}>
  apply(simp only:ran_def)
  using grid_image by force
```

The lemma `grid_image_approx` shows that the output of the classification is never ambiguous (i.e., two or more classification output having the value 1).

```
lemma grid_dom': <dom (predict_seq_layer_m' grid.NeuralNet) = {a. length a = 4}>
  unfolding predict_seq_layer_m'_def predict_seq_layer_m_def
  by (smt (verit, best) Collect_cong dom_def grid_img_defined' predict_seq_layer_m'_def predict_seq_layer_m_def)
```

```
lemma grid_dom_mat': <dom (λ x . predict_seq_layer_m grid.NeuralNet (vec_of_list x)) = {a. length a = 4}>
  by (simp add: grid_defs predict_seq_layer_m_def activation_defs dom_def dim_col_mat_of_col_list
  dim_row_mat_of_col_list)
```

```
lemma <x3 ∈ {0.96..1.00} ∧ x2 ∈ {0.96..1.00}
  ∧ x1 ∈ {0.00..0.04} ∧ x0 ∈ {0.00..0.04} ⇒ predict_seq_layer_m' grid.NeuralNet [x3, x2, x1, x0] = Some [0, 1, 0]>
  by(simp add: Let_def)
```

```
lemma <x3 ∈ {0.00..0.04} ∧ x2 ∈ {0.00..0.04}
  ∧ x1 ∈ {0.96..1.00} ∧ x0 ∈ {0.96..1.00} ⇒ predict_seq_layer_m' grid.NeuralNet [x3, x2, x1, x0] = Some [0, 1, 0]>
  by(simp add: Let_def)
```

```
lemma <x3 ∈ {0.95..1.00} ∧ x2 ∈ {0.00..0.05}
  ∧ x1 ∈ {0.95..1.00} ∧ x0 ∈ {0.00..0.05} ⇒ predict_seq_layer_m' grid.NeuralNet [x3, x2, x1, x0] = Some [0, 0, 1]>
  by(simp add: Let_def)
```

```
lemma <x3 ∈ {0.00..0.1} ∧ x2 ∈ {0.96..1.00}
  ∧ x1 ∈ {0.00..0.1} ∧ x0 ∈ {0.96..1.00} ⇒ predict_seq_layer_m' grid.NeuralNet [x3, x2, x1, x0] = Some [0, 0, 1]>
  by(simp add: Let_def)
```

A common definition of safety in neural networks is the requirement that small changes to an input should not change the classification. For this grid example, we express such a verification goal as shown above, where we set a small bound of noise on the input, and verify that the output classification remains constant.

```
lemma grid_meets_predictions:
  <⊨il {inputs} (predict_seq_layer_m' grid.NeuralNet) {intervals_of_l 0.000001 predictions}>
  by(simp add: ensure_testdata_interval_list_def upper_Interval lower_Interval predictions_def
  intervals_of_l_def inputs_def in_set_interval)
```

```
lemma grid_meets_expectations_max_classifier:
  <⊨l {inputs_small} (predict_seq_layer_m' grid.NeuralNet) {expectations_small}>
  by (simp add: ensure_testdata_max_list_def expectations_small_def inputs_small_def)
```

```
lemma grid_min_delta_classifier:
  <1.0 ⊨ predict_seq_layer_m' grid.NeuralNet>
  unfolding ensure_delta_min_def Prediction_Utils.δ_min_def max_list_def
  using grid_image_approx
  by auto
```

The lemmas `grid_meets_predictions`, `grid_meets_expectations_max_classifier` and `grid_min_delta_classifier` show that our definition of the grid neural network computes the same prediction as the TensorFlow trained network.

```
end
```


Bibliography

- [1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: large-scale machine learning on heterogeneous systems, 2015. URL: <https://www.tensorflow.org/>. Software available from tensorflow.org.
- [2] C. C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. Springer Publishing Company, Incorporated, 1st edition, 2018. ISBN: 3319944622.
- [3] T. Bray, editor. The JavaScript Object Notation (JSON) Data Interchange Format. Online: <https://datatracker.ietf.org/doc/html/rfc8259>. Dec. 2017.
- [4] A. D. Brucker. Nano JSON. *Archive of Formal Proofs*, 2022. ISSN: 2150-914x. https://isa-afp.org/entries/Nano_JSON.html, Formal proof development.
- [5] A. D. Brucker and A. Stell. Verifying feedforward neural networks for classification in Isabelle/HOL. In M. Chechik, J.-P. Katoen, and M. Leucker, editors, *Formal Methods (FM 2023)*. Lübeck, Germany, 2023. ISBN: 978-3-642-38915-3. URL: <http://www.brucker.ch/bibliography/abstract/brucker.ea-feedforward-nn-verification-2023>.
- [6] BS EN 50128:2011: Railway applications – Communication, signalling and processing systems – Software for railway control and protecting systems. Apr. 2014.
- [7] Common Criteria for Information Technology Security Evaluation (Version 3.1, Release 5). Available at <https://www.commoncriteriaportal.org/cc/>. 2017.
- [8] M. Eberl. The Error Function. *Archive of Formal Proofs*, Feb. 2018. ISSN: 2150-914x. https://isa-afp.org/entries/Error_Function.html, Formal proof development.
- [9] ECMA-404: The JSON data interchange syntax. Online: <https://www.ecma-international.org/publications-and-standards/standards/ecma-404/>. Dec. 2017.
- [10] F. Haftmann and L. Bulwahn. Code generation from Isabelle/HOL theories, 2021. URL: <http://isabelle.in.tum.de/doc/codegen.pdf>.
- [11] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020. DOI: 10.1038/s41586-020-2649-2. URL: <https://doi.org/10.1038/s41586-020-2649-2>.
- [12] D. Matichuk, T. Murray, and M. Wenzel. Eisbach: a proof method language for Isabelle. *Journal of Automated Reasoning*, 56(3):261–282, Mar. 2016. DOI: 10.1007/s10817-015-9360-2.
- [13] L. Noschinski. Graph Theory. *Archive of Formal Proofs*, Apr. 2013. ISSN: 2150-914x. https://isa-afp.org/entries/Graph_Theory.html, Formal proof development.

- [14] D. Smilkov, N. Thorat, Y. Assogba, A. Yuan, N. Kreeger, P. Yu, K. Zhang, S. Cai, E. Nielsen, D. Soergel, S. Bileschi, M. Terry, C. Nicholson, S. N. Gupta, S. Sirajuddin, D. Sculley, R. Monga, G. Corrado, F. B. Viégas, and M. Wattenberg. Tensorflow.js: machine learning for the web and beyond. *CoRR*, abs/1901.05350, 2019. arXiv: 1901.05350. URL: <http://arxiv.org/abs/1901.05350>.
- [15] A. Stell. *Trustworthy Machine Learning for High-Assurance Systems*. PhD thesis, University of Exeter, Exeter, UK, 2025.