

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)	16
2.2.1	Encoder	16
2.2.2	Example Import	18
2.3	Common Infrastructure (NN_Common)	22
2.3.1	Utility Functions	22
2.3.2	Data Import	22
2.3.3	Common Infrastructure for Proof Tactics	22
3	Activation Functions	25
3.1	Defining Activation Functions and Their Derivatives (Activation_Functions)	25
3.1.1	Activation Functions	25
3.1.2	Derivatives of Activation Functions	29
3.1.3	Single Class Folding Activation Functions	29
3.1.4	Multiclass Folding Activation Functions	31
3.2	Encoding of Activation Functions (Activation_Functions)	32
4	Neural Networks as Directed Graphs	33
4.1	Useful Definitions for Analyzing Predictions (Prediction_Utils)	33
4.2	Desirable Properties of Neural Networks Predictions (Properties)	37
4.2.1	Approximate Comparison of Results	37
4.2.2	Maximum Classifiers	38
4.2.3	Distance-based Properties	39
4.3	Neural Networks as Graphs (NN_Digraph)	42
4.3.1	Neurons as Vertices	44
4.3.2	Arcs (Edges)	44
4.3.3	Updating Neurons	45
4.3.4	Updating arcs (edges)	46
4.3.5	The empty neural network	49
4.3.6	Computing Predictions of Neural Networks	49
4.4	Main Theory (Digraph) (NN_Digraph_Main)	50
5	Neural Networks as Layers	53
5.1	Preliminaries	53
5.1.1	Useful Definitions for Analysing Matrix Predictions (Prediction_Utils_Matrix)	53
5.1.2	Desirable Properties of Neural Networks Predictions (Properties_Matrix)	55
5.1.3	Sequential Layers (NN_Layers)	57

5.1.4	Neural Network Lipschitz Continuity	59
5.2	Models	69
5.2.1	Digraphs as Layers (📄NN_Digraph_Layers)	69
5.2.2	Neural Network as Sequential Layers using Lists (📄NN_Layers_List_Main)	76
5.2.3	Neural Network as Sequential Layers using Vector Spaces (📄NN_Layers_Matrix_Main)	82
5.3	Main Theory (Layers) (📄NN_Layers_Main)	86
5.3.1	Converting between List-based and Matrix-based Sequential Layer Models	86
5.3.2	Converting Between List/Matrix-based Representations Preserves Consistency	87
5.3.3	Semantic Equivalence of List-based and Matrix-based Models	91
6	Main Theory Including all Model Types (📄NN_Main)	95
7	Reference Manual (thy)	97
7.1	Importing Neural Networks and Data (📄NN_Manual)	97
7.2	Proof Methods (📄NN_Manual)	98
8	Examples	99
8.1	Compass	99
8.1.1	Neural Networks as Directed Graphs (📄Compass_Digraph)	99
8.1.2	Neural Networks as List of Layers using List Types (📄Compass_Layers_List)	103
8.1.3	Neural Networks as List of Layers using Matrix Types (📄Compass_Layers_Matrix)	106
8.2	Line Classification Model (📄Grid_Layers) (📄Grid_Layers)	109
8.2.1	Layer-based Modelling using List Types(📄Grid_Layers_List)	110
8.2.2	Layer-based Modelling using List Types (📄Grid_Layers_Matrix)	113

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.

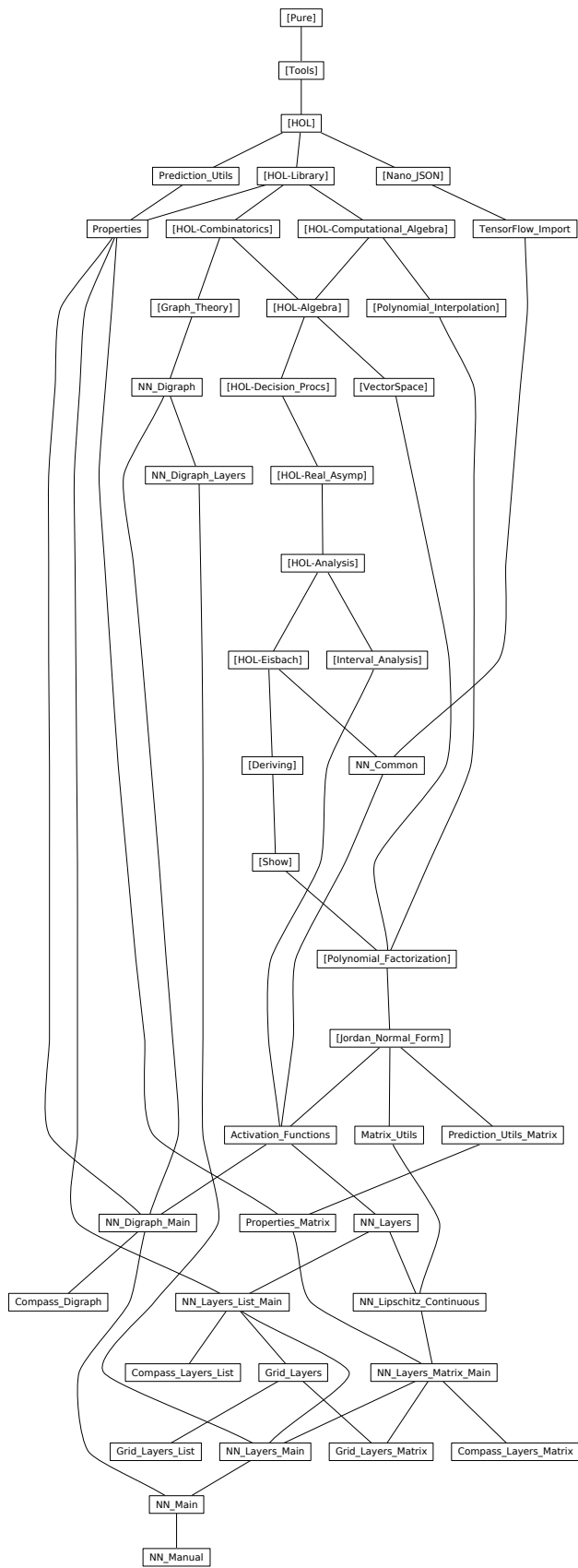


Figure 1.1: The Dependency Graph of the Isabelle Theories.

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.. $\text{length}$  xs]>
  <proof>
```

```
lemma map2_to_map_idx:
  <(map2 (*) xs (ys)) = map ( $\lambda$  i. xs!i * ys!i) [0.. $\text{min}$  (length xs) (length ys)]>
  <proof>
```

```
lemma map2_mult_commute:
  <map2 (*) (xs::'a::comm_ring list) ys = map2 (*) ys xs>
  <proof>
```

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>
  <proof>
```

```
lemma index_mult_vec_mat: <i < dim_col A  $\implies$  (v v * A) $ i = col A i  $\cdot$  v>
  <proof>
```

```
lemma dim_col_mat_list: < $\forall$  m  $\in$  set (mat_to_list M). dim_col M = length m >
  <proof>
```

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

```
lemma scalar_prod_list:
  <((vec_of_list v)  $\cdot$  (vec_of_list w)) = ( $\sum$  i  $\in$  {0.. $\text{length}$  w}. v!i * w!i)>
```

<proof>

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

<proof>

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

<proof>

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

<proof>

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

<proof>

lemma *vec_of_list_ext*: $\langle \text{vec_of_list } xs = \text{vec_of_list } ys \implies xs = ys \rangle$

<proof>

lemma *list_of_vec_ext*: $\langle \text{list_of_vec } xs = \text{list_of_vec } ys \implies xs = ys \rangle$

<proof>

lemma *map_if_lam*:

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

<proof>

lemma *map_if_lam'*:

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

<proof>

lemma *map_if_lam''*:

$\langle \text{map } (\lambda i. \text{map } (\lambda ia. \text{if } i < n \text{ then } P i ia \text{ else } Q i ia) [0..<m]) [0..<n] \rangle$

$= \text{map } (\lambda i. \text{map } (\lambda ia. P i ia) [0..<m]) [0..<n] \rangle$

<proof>

lemma *vec_add_list*:

assumes $\langle \text{length } v = \text{length } w \rangle$

shows $\langle \text{list_of_vec } ((\text{vec_of_list } v) + (\text{vec_of_list } w)) = \text{map2 } (+) v w \rangle$

<proof>

lemma *vec_add_list'*:

assumes $\langle \text{length } v = \text{length } w \rangle$

shows $\langle ((\text{vec_of_list } v) + (\text{vec_of_list } w)) = \text{vec_of_list } (\text{map2 } (+) v w) \rangle$

<proof>

lemma *mat_col_list*:

assumes $\langle i < \text{length As} \rangle$

and $\langle \forall a \in \text{set As}. \forall a' \in \text{set As}. \text{length } a = \text{length } a' \wedge a \neq [] \rangle$

and $\langle d = \text{length } (\text{hd As}) \rangle$

shows $\langle \text{list_of_vec } (\text{col } (\text{mat_of_cols_list } d \text{ As}) i) = \text{As!}i \rangle$

<proof>

lemma *mult_vec_mat_col_list*:

assumes $\langle \text{length } vs = n \rangle$

and $\langle \forall a \in \text{set As}. \forall a' \in \text{set As}. \text{length } a = \text{length } a' \wedge a \neq [] \rangle$

and $\langle \text{length } (\text{hd As}) = d \rangle$

and $\langle \text{length } As = n \rangle$
and $\langle As \neq [] \rangle$
shows $\langle \text{list_of_vec } ((\text{vec_of_list } vs) \cdot v * (\text{mat_of_cols_list } d \ As)) = \text{map } (\lambda i. \sum ia = 0..<\text{length } vs. As ! i ! ia * vs ! ia)$
 $[0..<n] \rangle$
 $\langle \text{proof} \rangle$

lemma *mult_vec_mat_row_list*:

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

lemma *mult_vec_mat_row_list'*:

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

lemma *col_of_rows_list*:

assumes $\langle d = \text{Min } (\text{set } (\text{map } \text{length } As)) \rangle$
and $\langle i < d \rangle$
shows $\langle \text{list_of_vec } (\text{col } (\text{mat_of_rows_list } d \ As) \ i) = \text{map } (\lambda as. (as ! i)) \ As \rangle$
 $\langle \text{proof} \rangle$

lemma *col_of_rows_list'*:

assumes $\langle \forall as \in \text{set } As. \text{length } as = d \rangle$
and $\langle As \neq [] \rangle$
shows $\langle (\text{col } (\text{mat_of_rows_list } d \ As) \ i) = \text{vec_of_list } (\text{map } (\lambda as. (as ! i)) \ As) \rangle$
 $\langle \text{proof} \rangle$

lemma *list_mat*: $\langle \text{mat_of_rows_list } (\text{dim_col } A) (\text{mat_to_list } A) = A \rangle$

$\langle \text{proof} \rangle$

lemma *list_mat_transpose_transpose*: $\langle (\text{mat_of_rows_list } (\text{dim_col } x^T) (\text{mat_to_list } x^T))^T = x \rangle$

$\langle \text{proof} \rangle$

lemma *mat_list*:

$\langle \forall r \in \text{set}(rs). \text{length } r = \text{dimc} \implies \text{mat_to_list } (\text{mat_of_rows_list } \text{dimc } rs) = rs \rangle$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

```

lemma scalar_prod_sum_list_lv_eq:
  assumes same_dim: <dim_vec (x::'a::comm_ring Matrix.vec) = dim_vec y>
  shows <x · y ≡ sum_list (map2 (*) (list_of_vec x) (list_of_vec y))>
  <proof>

```

```

lemma scalar_prod_sum_list_vl_eq:
  assumes same_dim: <length (x::'a::comm_ring list) = length y>
  shows <(vec_of_list x) · (vec_of_list y) ≡ sum_list (map2 (*) x y)>
  <proof>

```

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>

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>

TensorFlow.js does export a neural network in a format consisting out of a JSON file 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 `TensorFlowJson:TensorFlowJson` provides, foremost, a function for parsing the JSON exported neural network in the format supported by TensorFlow. This function, `TensorFlowJson.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 `NanoJsonParser.json_of_string`, which is provided by Nano JSON [4]).

The function `TensorFlowJson.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 `TensorFlowJson.transform_json` results in the following transformed `weightsManifest`:

```

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

```

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⟩

We use the mechanism of attaching a symtab 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 symtab is registered as follows:

⟨ML⟩

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 / 10000000) ],
          ARRAY
          [ NUMBER (- 12952799 / 10000000), NUMBER (3662836 / 10000000),
            NUMBER (9530481 / 10000000) ],
          ARRAY
          [ NUMBER (- 2857958 / 10000000), NUMBER (6922799 / 10000000),
            NUMBER (35006753 / 10000000) ],
          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) ] ] ] ],
OBJECT
  [ ("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>

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

2.3.2 Data Import

<ML>

2.3.3 Common Infrastructure for Proof Tactics

<ML>

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
  <proof>
```

3.1.1 Activation Functions

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

lemma identity_linear[simp]: <affine_fun identity>
  <proof>

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 $\langle \text{sigmoid} = (\text{logistic} (1.0::\text{real}) 1.0) \rangle$
 $\langle \text{proof} \rangle$

lemma $\langle \text{sigmoid}_{\text{taylor}} n = (\text{logistic}_{\text{taylor}} n (1.0::\text{real}) 1.0) \rangle$
 $\langle \text{proof} \rangle$

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 $\langle \text{relu} = (\text{generalized_relu} (0.0::\text{real}) \text{None} (0.0)) \rangle$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

definition $\text{softmax}_{\text{taylor}} :: \text{nat} \Rightarrow ('a :: \{\text{banach, real_normed_algebra_1, inverse}\}) \text{ list} \Rightarrow 'a \text{ list where}$

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

definition $\text{msoftmax}_{\text{taylor}} :: \text{nat} \Rightarrow ('a :: \{\text{banach, real_normed_algebra_1, inverse}\}) \text{ vec} \Rightarrow 'a \text{ vec where}$

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

lemma $\text{softmax}_{\text{taylor}2}$:

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

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

$\langle \text{proof} \rangle$

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*:

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

<proof>

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

<proof>

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

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

<proof>

definition *logistic' L k v₀* = $(\lambda\ v. (\exp\ ((-k)*(v-v_0)) * k * L) / (1 + \exp\ ((-k)*(v-v_0)))^2)$

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

<proof>

definition *tanh'* = $(\lambda\ v. 1 - ((\tanh\ v)^2))$

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

<proof>

definition *softplus'* = $(\lambda\ v. 1 / (1 + \exp(-v)))$

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

<proof>

definition *prelu1'* = $(\lambda\ v. 1)$

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

<proof>

definition *silu'* = $(\lambda\ v. (1 + \exp(-v) + v * (\exp(-v))) / ((1 + \exp(-v))^2))$

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

<proof>

definition *gaussian'* = $(\lambda\ v. -2 * v * \exp(-v^2))$

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

<proof>

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* | *Sigmoid_{taylor} nat* | *ReLU* | *GReLU real <real option> real*
 | *Softplus* | *SoftSign* | *Swish* | *Swish_{taylor} nat* | *GeLUapprox* | *ELU real*
 | *ELU_{taylor} nat real* | *SELU* | *SELU_{taylor} nat* | *PReLU real* | *SiLU* | *SiLU_{taylor} nat*
 | *Gaussian* | *Gaussian_{taylor} nat* | *Exp* | *Exp_{taylor} nat* | *HardSigmoid*

fun φ_{single} : $\langle \text{activation}_{\text{single}} \Rightarrow (\text{real} \Rightarrow \text{real}) \text{ option} \rangle$ **where**

$\langle \varphi_{\text{single}} \text{ Identity} = \text{Some identity} \rangle$
 $\langle \varphi_{\text{single}} \text{ Sign} = \text{Some sign} \rangle$
 $\langle \varphi_{\text{single}} \text{ BinaryStep} = \text{Some binary_step} \rangle$
 $\langle \varphi_{\text{single}} \text{ SoftSign} = \text{Some softsign} \rangle$
 $\langle \varphi_{\text{single}} (\text{Logistic } L\ k\ v_0) = \text{Some } (\text{logistic } L\ k\ v_0) \rangle$
 $\langle \varphi_{\text{single}} (\text{Logistic}_{\text{taylor}}\ n\ L\ k\ v_0) = \text{Some } (\text{logistic}_{\text{taylor}}\ n\ L\ k\ v_0) \rangle$
 $\langle \varphi_{\text{single}} \text{ Sigmoid} = \text{Some sigmoid} \rangle$

```

|⟨φsingle (Sigmoidtaylor n) = Some (sigmoidtaylor n)⟩
|⟨φsingle Swish = Some swish⟩
|⟨φsingle (Swishtaylor n) = Some (swishtaylor n)⟩
|⟨φsingle Tanh = Some tanh⟩
|⟨φsingle ReLU = Some relu⟩
|⟨φsingle GeLUapprox = Some gelu_approx⟩
|⟨φsingle (GReLU α m t) = Some (generalized_relu α m t)⟩
|⟨φsingle Softplus = Some softplus⟩
|⟨φsingle (ELU α) = Some (elu α)⟩
|⟨φsingle (ELUtaylor n α) = Some (elutaylor n α)⟩
|⟨φsingle SELU = Some selu⟩
|⟨φsingle (SELUtaylor n) = Some (selutaylor n)⟩
|⟨φsingle Exp = Some exp⟩
|⟨φsingle (Exptaylor n) = Some (exptaylor n)⟩
|⟨φsingle HardSigmoid = Some hard_sigmoid⟩
|⟨φsingle (PReLU α) = Some (prelu α)⟩
|⟨φsingle SiLU = Some silu⟩
|⟨φsingle (SiLUtaylor n) = Some (silutaylor n)⟩
|⟨φsingle Gaussian = Some gaussian⟩
|⟨φsingle (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 `φsingle` provides easy access to the activation function itself.

```

fun φsingle ' :: ⟨activationsingle ⇒ (real ⇒ real option)⟩ where
  ⟨φsingle ' Identity = (λv. Some (identity' v))⟩
| ⟨φsingle ' Sign = (λv. None)⟩
| ⟨φsingle ' BinaryStep = (λv. None)⟩
| ⟨φsingle ' (Logistic L k v0) = (λv. Some (logistic' L k v0 v))⟩
| ⟨φsingle ' (Logistictaylor n L k v0) = (λv. None)⟩
| ⟨φsingle ' Tanh = (λv. Some (tanh' v))⟩
| ⟨φsingle ' ReLU = (λv. None)⟩
| ⟨φsingle ' Softplus = (λv. Some (softplus' v))⟩
| ⟨φsingle ' (ELU α) = (λv. None)⟩
| ⟨φsingle ' (ELUtaylor n α) = (λv. None)⟩
| ⟨φsingle ' (PReLU α) = (λ v. if α = 1 then Some (prelu1' v) else None)⟩
| ⟨φsingle ' SiLU = (λv. Some (silu' v))⟩
| ⟨φsingle ' (SiLUtaylor n) = (λv. None)⟩
| ⟨φsingle ' Gaussian = (λv. Some (gaussian' v))⟩
| ⟨φsingle ' (Gaussiantaylor n) = (λv. None)⟩
| ⟨φsingle ' (GReLU v va vb) = (λv. 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 `φsingle '` defines, for derivable activation functions, their derivative. Note that we require deriv-

ability 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)))>
<proof>

```

3.1.4 Multiclass Folding Activation Functions

datatype *activation_{multi}* = *mIdentity* | *mSign* | *mBinaryStep* | *mLogistic real real real* | *mLogistic_{taylor} 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 *activation_{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 Activion Functions (Activation_Functions)

$\langle ML \rangle$

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** $\text{max}_{list} :: \langle 'a::\text{linorder list} \Rightarrow 'a \rangle$ where
 $\langle \text{max}_{list} = \text{Max o set} \rangle$

definition $\text{min}_{list} :: \langle 'a::\text{linorder list} \Rightarrow 'a \rangle$ where
 $\langle \text{min}_{list} = \text{Min o set} \rangle$

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

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

lemma $\text{max}_{list_append_eq} : \langle \text{max}_{list} (xs @ [x]) = \text{max}_{list} xs \vee \text{max}_{list} (xs @ [x]) = x \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{min}_{list_append_eq} : \langle \text{min}_{list} (xs @ [x]) = \text{min}_{list} xs \vee \text{min}_{list} (xs @ [x]) = x \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{max}_{list_cons_eq} : \langle \text{max}_{list} (x\#xs) = \text{max}_{list} xs \vee \text{max}_{list} (x\#xs) = x \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{min}_{list_cons_eq} : \langle \text{min}_{list} (x\#xs) = \text{min}_{list} xs \vee \text{min}_{list} (x\#xs) = x \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{max}_{list_append_limit} : \text{assumes } \langle xs \neq [] \rangle \text{ shows } \langle \text{max}_{list} xs \leq \text{max}_{list} (xs @ [x]) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{min}_{list_append_limit} : \text{assumes } \langle xs \neq [] \rangle \text{ shows } \langle \text{min}_{list} (xs @ [x]) \leq \text{min}_{list} xs \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{max}_{list_cons_limit} : \text{assumes } \langle xs \neq [] \rangle \text{ shows } \langle \text{max}_{list} xs \leq \text{max}_{list} (x\#xs) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{min}_{list_cons_limit} : \text{assumes } \langle xs \neq [] \rangle \text{ shows } \langle \text{min}_{list} (x\#xs) \leq \text{min}_{list} xs \rangle$
 $\langle \text{proof} \rangle$

Converting Predictions to Percentages **definition** $\text{prediction2percentage} :: \langle \text{real list} \Rightarrow \text{real list} \rangle$ where
 $\langle \text{prediction2percentage } l = (\text{let } m = \text{max}_{list} l \text{ in } \text{map } (\lambda e. e / m * 100.0) l) \rangle$

lemma $\text{prediction2percentage_is_percentage} :$
assumes $\langle 0 < \text{max}_{list} l \rangle$
shows $\langle \forall x \in \text{set } (\text{prediction2percentage } l). x \leq 100.0 \rangle$
 $\langle \text{proof} \rangle$

lemma prediction2percentage_id: **assumes** $\langle \max_{list} p = 100 \rangle$ **shows** $\langle \text{prediction2percentage } p = p \rangle$
 $\langle \text{proof} \rangle$

lemma prediction2percentage_min_id:
assumes $\langle 0 < \max_{list} p \rangle$
shows $\langle (0 \leq \min_{list} (\text{prediction2percentage } p)) = (0 \leq \min_{list} p) \rangle$
 $\langle \text{proof} \rangle$

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

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

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

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

Minimum Prediction **definition** $\text{posmin_of} :: \langle 'a::\text{linorder list} \Rightarrow (\text{nat} \times 'a) \text{ option} \rangle$ **where**
 $\langle \text{posmin_of } l = (\text{let } m = \min_{list} l \text{ in find } (\lambda e. \text{snd } e = m) (\text{enumerate } o \ l)) \rangle$

definition $\text{pos_of_min} :: \langle 'a::\text{linorder list} \Rightarrow \text{nat option} \rangle$ **where**
 $\langle \text{pos_of_min } l = \text{map_option fst } (\text{posmin_of } l) \rangle$

definition $\text{posmin_of}' :: \langle 'a::\text{linorder list} \Rightarrow (\text{nat} \times 'a) \text{ option} \rangle$ **where**
 $\langle \text{posmin_of}' l = (\text{if } l = [] \text{ then None else Some } ((\text{hd } o \ \text{rev } o (\text{sort_key snd } o (\text{enumerate } o)) \ l))) \rangle$

definition $\text{pos_of_min}' :: \langle 'a::\text{linorder list} \Rightarrow \text{nat option} \rangle$ **where**
 $\langle \text{pos_of_min}' l = \text{map_option fst } (\text{posmin_of}' l) \rangle$

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

lemma posmax_of_split: $\langle \text{posmax_of } (xs @ [x]) = \text{posmax_of } (xs) \vee \text{posmax_of } (xs @ [x]) = \text{Some } (\text{length } xs, x) \rangle$
 $\langle \text{proof} \rangle$

lemma posmin_of_split: $\langle \text{posmin_of } (xs @ [x]) = \text{posmin_of } (xs) \vee \text{posmin_of } (xs @ [x]) = \text{Some } (\text{length } xs, x) \rangle$
 $\langle \text{proof} \rangle$

lemma pos_of_max_split:
 $\langle \text{pos_of_max } (xs @ [x]) = \text{pos_of_max } (xs) \vee \text{pos_of_max } (xs @ [x]) = \text{Some } (\text{length } xs) \rangle$
 $\langle \text{proof} \rangle$

lemma pos_of_min_split:
 $\langle \text{pos_of_min } (xs @ [x]) = \text{pos_of_min } (xs) \vee \text{pos_of_min } (xs @ [x]) = \text{Some } (\text{length } xs) \rangle$
 $\langle \text{proof} \rangle$

lemma posmax_of_none: $\langle (\text{posmax_of } xs = \text{None}) = (xs = []) \rangle$
 $\langle \text{proof} \rangle$

lemma posmin_of_none: $\langle (\text{posmin_of } xs = \text{None}) = (xs = []) \rangle$
 $\langle \text{proof} \rangle$

lemma posmax_of_in_snd: $\langle (\text{posmax_of } xs) = \text{Some } p \implies \text{snd } p \in \text{set } xs \rangle$

<proof>

lemma *posmin_of_in_snd*: $\langle (\text{posmin_of } xs) = \text{Some } p \implies \text{snd } p \in \text{set } xs \rangle$
<proof>

lemma *pos_of_max_none*: $\langle (\text{pos_of_max } xs = \text{None}) = (xs = []) \rangle$
<proof>

lemma *pos_of_min_none*: $\langle (\text{pos_of_min } xs = \text{None}) = (xs = []) \rangle$
<proof>

lemma *take_nth_drop_eq*:
assumes $\langle xs \neq [] \rangle$
and $\langle n < \text{length } xs \rangle$
shows $\langle xs = ((\text{take } n \text{ } xs)@[xs!n]@(\text{drop } (n+1) \text{ } xs)) \rangle$
<proof>

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 $\langle \text{Max } (\text{set } ((\text{take } n \text{ } xs)@[xs!n]@(\text{drop } (n+1) \text{ } xs))) = ((\text{take } n \text{ } xs)@[xs!n]@(\text{drop } (n+1) \text{ } xs))!n \rangle$
<proof>

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 $\langle \text{Min } (\text{set } ((\text{take } n \text{ } xs)@[xs!n]@(\text{drop } (n+1) \text{ } xs))) = ((\text{take } n \text{ } xs)@[xs!n]@(\text{drop } (n+1) \text{ } xs))!n \rangle$
<proof>

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 $\langle \text{Max } (\text{set } xs) = xs!n \rangle$
<proof>

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 $\langle \text{Min } (\text{set } xs) = xs!n \rangle$
<proof>

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

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⟩

lemma nth_higher_min:

assumes ⟨xs ≠ []⟩
and ⟨n < length xs⟩
and ⟨∀ x ∈ set ((take n xs)@(drop (n+1) xs)). x > xs!n⟩
shows ⟨∀ j < n. xs!j > xs!n⟩
⟨proof⟩

lemma posmax_of_le:

assumes ⟨xs ≠ []⟩
and ⟨n < length xs⟩
and ⟨∀ x ∈ set ((take n xs)@(drop (n+1) xs)). x < xs!n⟩
shows ⟨posmax_of xs = Some (n,xs!n)⟩
⟨proof⟩

lemma posmin_of_le:

assumes ⟨xs ≠ []⟩
and ⟨n < length xs⟩
and ⟨∀ x ∈ set ((take n xs)@(drop (n+1) xs)). x > xs!n⟩
shows ⟨posmin_of xs = Some (n,xs!n)⟩
⟨proof⟩

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)⟩
⟨proof⟩

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)⟩
⟨proof⟩

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

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⟩

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⟩

lemma le_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 $\langle c_1 * b + c_0 < c_1' * b + c_0' \rangle$
 $\langle proof \rangle$

lemma *le_linear_real'*:

assumes *b_bound*: $\langle (b::real) \in \{lb..up\} \rangle$
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 $\langle c_1 * b + c_0 < c_1' * b + c_0' \rangle$
 $\langle proof \rangle$

lemma *pos_max_leq'*: $\langle (pos_of_max\ xs = Some\ n) \implies \forall x \in set\ xs.\ x \leq xs!n \rangle$
 $\langle proof \rangle$

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 $\langle approx\ a\ \varepsilon\ b = (|a-b| \leq \varepsilon) \rangle$

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

fun *checkget_result_list* **where**

$\langle checkget_result_list_ _ None\ None = (None, True) \rangle$
 $| \langle checkget_result_list\ \varepsilon\ (Some\ xs)\ (Some\ ys) = (Some\ xs,\ fold\ (\wedge)\ (map2\ (\lambda\ x\ y.\ x \approx[\varepsilon] \approx y)\ xs\ ys)\ True) \rangle$
 $| \langle checkget_result_list_ _ r_ = (r, False) \rangle$

definition $\langle check_result_list\ r\ \varepsilon\ s = snd\ (checkget_result_list\ \varepsilon\ r\ s) \rangle$

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

fun *checkget_result_singleton* **where**

$\langle checkget_result_singleton_ _ None\ None = (None, True) \rangle$
 $| \langle checkget_result_singleton\ \varepsilon\ (Some\ x)\ (Some\ y) = (Some\ x,\ x \approx[\varepsilon] \approx y) \rangle$
 $| \langle checkget_result_singleton_ _ r_ = (r, False) \rangle$

definition $\langle check_result_singleton\ r\ \varepsilon\ s = snd\ (checkget_result_singleton\ \varepsilon\ r\ s) \rangle$

notation *check_result_singleton* $((_)/ \approx[_] \approx_s (_))$ [60, 60] 60)

definition

ensure_testdata_range_list :: $\langle real \Rightarrow real\ list\ list \Rightarrow (real\ list \rightarrow real\ list) \Rightarrow real\ list\ list \Rightarrow bool \rangle$

where

$\langle ensure_testdata_range_list\ delta\ inputs\ P\ outputs$
 $= foldl\ (\wedge)\ True$
 $(map\ (\lambda\ e.\ (P\ (fst\ e)) \approx[\delta] \approx_l Some\ (snd\ e))$
 $(zip\ inputs\ outputs)) \rangle$

notation *ensure_testdata_range_list* $((_)\models_l \{(_)\} (_)\{(_)\})$ [61, 3, 90, 3] 60)

Interval Arithmetic

definition `interval_distance` :: $\langle 'a :: \{\text{preorder}, \text{minus}, \text{zero}, \text{ord}\} \text{ interval} \Rightarrow 'a \text{ interval} \Rightarrow 'a \rangle$ **where**
 `$\langle \text{interval_distance } a \ b = (\text{let } (la, ua) = \text{bounds_of_interval } a;$
 $(lb, ub) = \text{bounds_of_interval } b$
 $\text{in if } ua \leq lb \text{ then } lb - ua$
 $\text{else if } ub \leq la \text{ then } la - ub$
 $\text{else } 0 \rangle$`

fun `intervals_of_list` **where**
 `$\langle \text{intervals_of_list } [] = [] \rangle$
 $\langle \text{intervals_of_list } \delta (x \# xs) = (\text{Interval } (x - |\delta|, x + |\delta|)) \# (\text{intervals_of_list } \delta xs) \rangle$`

definition `$\langle \text{intervals_of_l } \delta = \text{map } (\text{intervals_of_list } \delta) \rangle$`

lemma `$\text{interval_in_implies_set}: (x \in \{a..b\}) \implies (x \in \text{set_of } (\text{Interval } (a,b)))$`
 `$\langle \text{proof} \rangle$`

lemma `$\text{in_set_interval}: a \leq b \implies (x \in \text{set_of } (\text{Interval } (a,b))) = (x \in \{a..b\})$`
 `$\langle \text{proof} \rangle$`

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

notation `$\text{check_result_list_interval_list } ((_)/ \approx_l (_)) [60, 60] 60$`

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

notation `$\text{ensure_testdata_interval_list } (\models_{il} \{(_)\} (_) \{(_)\} [3, 90, 3] 60)$`

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

(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* :: $\langle \text{real} \Rightarrow (\text{real list} \rightarrow \text{real list}) \Rightarrow \text{bool} \rangle$ **where**

$\langle \text{ensure_delta_min } \delta P = (\forall xs \in \text{ran } P. \delta \leq \delta_{min} xs) \rangle$

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

lemma *ensure_delta_min_dom*: $\langle \text{ensure_delta_min } \delta P = (\forall x \in \text{dom } P. \delta \leq \delta_{min} (\text{the } (P x))) \rangle$

$\langle \text{proof} \rangle$

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*:: $\langle 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow ('b::\{\text{linordered_ab_group_add}\}) \rangle$

assumes *identity*: $\langle \llbracket \text{length } x = \text{length } y \rrbracket \Longrightarrow (d x y = 0) = (x = y) \rangle$

and *symmetry*: $\langle (d x y = d y x) \rangle$

and *triangle_inequality*: $\langle \llbracket \text{length } x = \text{length } y ; \text{length } z = \text{length } y \rrbracket \Longrightarrow (d x z \leq d x y + d y z) \rangle$

begin

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

$\langle \text{proof} \rangle$

lemma $\langle \llbracket \text{length } x = \text{length } y ; \text{length } z = \text{length } y \rrbracket \Longrightarrow d x y + d y x \geq d x x \rangle$

$\langle \text{proof} \rangle$

lemma $\langle \text{length } x = \text{length } y \Longrightarrow 0 \leq d x y \rangle$

$\langle \text{proof} \rangle$

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**
 $\langle \text{mapfoldr } \text{map_f } \text{fold_f } e \text{ xs ys} = \text{foldr } \text{fold_f} (\text{map2 } (\lambda e_0 e_1 . \text{map_f } e_0 e_1) \text{ xs ys}) e \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma hamming_unroll: $\llbracket \text{length } xs = \text{length } ys \rrbracket$
 $\implies \text{hamming } (x\#xs) (y\#ys) = (\text{if } x = y \text{ then hamming } xs \text{ } ys \text{ else } 1 + \text{hamming } xs \text{ } ys)$
 $\langle \text{proof} \rangle$

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

global_interpretation hamming_distance: *distance hamming*
 $\langle \text{proof} \rangle$

definition manhattan:: $\langle \text{real list} \Rightarrow \text{real list} \Rightarrow \text{real} \rangle$ **where**
 $\langle \text{manhattan} = \text{mapfoldr } (\lambda x y . |x - y|) (+) 0 \rangle$

lemma manhattan_unroll: $\llbracket \text{length } xs = \text{length } ys \rrbracket$
 $\implies \text{manhattan } (x\#xs) (y\#ys) = |x - y| + \text{manhattan } xs \text{ } ys$
 $\langle \text{proof} \rangle$

lemma manhattan_positive: $\langle \text{length } x = \text{length } y \implies 0 \leq \text{manhattan } x \text{ } y \rangle$
 $\langle \text{proof} \rangle$

lemma manhattan_identity: $\langle \text{length } x = \text{length } y \implies (\text{manhattan } x \text{ } y = 0) = (x = y) \rangle$
 $\langle \text{proof} \rangle$

lemma manhattan_symmetry: $\langle \text{manhattan } x \text{ } y = \text{manhattan } y \text{ } x \rangle$
 $\langle \text{proof} \rangle$

lemma manhattan_triangle_inequality:
 $\langle \llbracket \text{length } xs = \text{length } ys ; \text{length } ys = \text{length } (zs::\text{real list}) \rrbracket$
 $\implies \text{manhattan } xs \text{ } zs \leq \text{manhattan } xs \text{ } ys + (\text{manhattan } ys \text{ } zs)$
 $\langle \text{proof} \rangle$

global_interpretation manhattan_distance: *distance manhattan*
 $\langle \text{proof} \rangle$

definition avg_difference:: $\langle \text{real list} \Rightarrow \text{real list} \Rightarrow \text{real} \rangle$ **where**
 $\langle \text{avg_difference } xs \text{ } ys = (\text{manhattan } xs \text{ } ys) / (\text{min } (\text{length } xs) (\text{length } ys)) \rangle$

global_interpretation avg_difference_distance: *distance avg_difference*
 $\langle \text{proof} \rangle$

definition euclidean:: $\langle \text{real list} \Rightarrow \text{real list} \Rightarrow \text{real} \rangle$ **where**
 $\langle \text{euclidean } X Y = \text{sqrt } (\text{mapfoldr } (\lambda x y . (x - y)^2) (+) 0 X Y) \rangle$

lemma euclidean_positive: $\langle \text{length } x = \text{length } y \implies 0 \leq \text{euclidean } x \text{ } y \rangle$
 $\langle \text{proof} \rangle$

lemma euclidean_identity: $\langle \text{length } x = \text{length } y \implies (\text{euclidean } x \text{ } y = 0) = (x = y) \rangle$
 $\langle \text{proof} \rangle$

lemma euclidean_symmetry: $\langle \text{euclidean } x \ y = \text{euclidean } y \ x \rangle$
 $\langle \text{proof} \rangle$

definition

$\text{check} :: \langle ('a \ \text{list} \Rightarrow 'a \ \text{list} \Rightarrow 'b) \Rightarrow ('b \Rightarrow \text{bool}) \Rightarrow 'a \ \text{list} \Rightarrow ('a \ \text{list} \rightarrow 'a \ \text{list}) \Rightarrow ('a \ \text{list} \ \text{option} \Rightarrow 'a \ \text{list} \ \text{option} \Rightarrow \text{bool}) \Rightarrow \text{bool} \rangle$ **where**
 $\langle \text{check } d \ P \ \text{input}_{ref} \ \text{prediction } P' \rangle$
 $= (\forall x \in \text{dom } \text{prediction}. P(d \ \text{input}_{ref} \ x) \longrightarrow P'(\text{prediction } x) (\text{prediction } \text{input}_{ref})) \rangle$

lemma $((\forall l \in \text{dom } \text{prediction}. P(\text{dist } i \ l) \longrightarrow P'(\text{prediction } l) (\text{prediction } i)))$
 $= ((\forall l \in \{l \in \text{dom } \text{prediction} . P(\text{dist } i \ l)\}. P'(\text{prediction } l) (\text{prediction } i)))$
 $\langle \text{proof} \rangle$

lemma hamming_update_1:

$\text{length } xs = \text{length } ys \Longrightarrow \text{hamming } xs \ ys \leq 1 \Longrightarrow (\exists i. xs = ys[i := xs!i])$
 $\langle \text{proof} \rangle$

lemma hamming_cases1:

assumes $l: \langle \text{length } xs = \text{length } ys \rangle$
and $h: \langle \text{hamming } xs \ ys \leq 1 \rangle$
and $p: \langle P \ xs \rangle$
and $u: \langle \bigwedge i. i < \text{length } xs \wedge ys = xs[i := (ys!i)] \Longrightarrow P \ ys \rangle$
shows $\langle P \ ys \rangle$
 $\langle \text{proof} \rangle$

lemma hamming_update_2:

$\text{length } xs = \text{length } ys \Longrightarrow \text{hamming } xs \ ys \leq 2 \Longrightarrow (\exists i \ j. xs = (ys[i := xs!i])[j := xs!j])$
 $\langle \text{proof} \rangle$

lemma hamming_cases2:

assumes $l: \langle \text{length } xs = \text{length } ys \rangle$
and $h: \langle \text{hamming } xs \ ys \leq 2 \rangle$
and $p: \langle P \ xs \rangle$
and $u: \langle \bigwedge i \ j. i < \text{length } xs \wedge j < \text{length } xs \wedge ys = xs[i := ys!i, j := ys!j] \Longrightarrow P \ ys \rangle$
shows $\langle P \ ys \rangle$
 $\langle \text{proof} \rangle$

lemma hamming_update_n:

$\text{length } xs = \text{length } ys \Longrightarrow \text{hamming } xs \ ys = \text{Suc } n \Longrightarrow (\exists i. \text{hamming } xs \ (ys[i := xs!i]) = n)$
 $\langle \text{proof} \rangle$

lemma hamming_update_3:

$\text{length } xs = \text{length } ys \Longrightarrow \text{hamming } xs \ ys \leq 3 \Longrightarrow (\exists i \ j \ k. xs = ys[i := xs!i, j := xs!j, k := xs!k])$
 $\langle \text{proof} \rangle$

lemma hamming_cases3:

assumes $l: \langle \text{length } xs = \text{length } ys \rangle$
and $h: \langle \text{hamming } xs \ ys \leq 3 \rangle$
and $p: \langle P \ xs \rangle$
and $u: \langle \bigwedge i \ j \ k. i < \text{length } xs \wedge j < \text{length } xs \wedge k < \text{length } xs \wedge ys = xs[i := ys!i, j := ys!j, k := ys!k] \Longrightarrow P \ ys \rangle$

shows <P ys>
<proof>

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 out 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 $\text{upd_neuron} :: \langle ('a, 'b) \text{nn_pregraph} \Rightarrow (('a, 'b) \text{Neuron} \Rightarrow ('a, 'b) \text{Neuron}) \Rightarrow ('a, 'b) \text{nn_pregraph} \rangle$ **where**
 $\langle \text{upd_neuron } G \text{ upd} = (\text{let } \text{upd_Neuron} = \text{case_neuron } \text{In } \text{Out } (\lambda n. \text{Neuron } (\text{upd } n))$
in $\langle \langle \langle \langle \langle \text{verts} = \text{upd_Neuron } '(\text{verts } G) ,$
 $\text{arcs} = (\lambda a. \langle \langle \omega = \omega a,$
 $\text{tl} = \text{upd_Neuron } (\text{tl } a),$
 $\text{hd} = \text{upd_Neuron } (\text{hd } a) \rangle \rangle) '(\text{arcs } G),$
 $\text{tail} = \text{tail } G,$
 $\text{head} = \text{head } G$
 $\rangle \rangle \rangle \rangle \rangle$

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

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

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

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 $\text{id_vert_inj} : \langle \text{inj_on } \text{uid } (\text{verts } G) \rangle$
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 $\text{nn_pregraph} : \text{nn_pregraph } G \langle \text{proof} \rangle$

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 nn_pregraph_add_neuron:
  assumes <uid n ∉ (uids G) ∨ n ∈ verts G >
  shows <nn_pregraph (add_vert n)>
  <proof>
```

```
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)>
  <proof>
end
```

The operation `pre_digraph.del_vert` preserves neural networks

```
context nn_pregraph
begin
```

```
lemma nn_pregraph_del_vert: <nn_pregraph (del_vert n)>
  <proof>
```

```
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)>
  <proof>
```

declare *add_nn_edge_def*[code]

lemma *nn_pregraph_add_nn_edge*: $\langle \text{nn_pregraph } (\text{add_nn_edge } G \ a) \rangle$
<proof>

The operation *del_arc* preserves neural networks

lemma *nn_pregraph_del_arc*: $\langle \text{nn_pregraph } (\text{del_arc } a) \rangle$
<proof>

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]: $\langle \beta \ n = \beta \ (\text{upd}_{\alpha} \ \alpha' \ n_{id} \ n) \rangle$
<proof>

lemma *wf_digraph_update_neuron*:
assumes $\langle \forall \ n. \ \text{Neuron.uid } n = \text{Neuron.uid } (\text{upd } n) \rangle$
shows $\langle \text{wf_digraph } (\text{upd_neuron } G \ \text{upd}) \rangle$
<proof>

lemma *fn_digraph_update_neuron*:
assumes $\langle \forall \ n. \ \text{Neuron.uid } n = \text{Neuron.uid } (\text{upd } n) \rangle$
shows $\langle \text{fn_digraph } (\text{upd_neuron } G \ \text{upd}) \rangle$
<proof>

lemma *nomulti_digraph_update_neuron*:
assumes $\langle \forall \ n. \ \text{Neuron.uid } n = \text{Neuron.uid } (\text{upd } n) \rangle$
shows $\langle \text{nomulti_digraph } (\text{upd_neuron } G \ \text{upd}) \rangle$
<proof>

lemma *loopfree_digraph_update_neuron*:
assumes $\langle \forall \ n. \ \text{Neuron.uid } n = \text{Neuron.uid } (\text{upd } n) \rangle$
shows $\langle \text{loopfree_digraph } (\text{upd_neuron } G \ \text{upd}) \rangle$
<proof>

lemma *nn_pregraph_update_neuron*:

assumes $\langle \forall n. \text{Neuron.oid } n = \text{Neuron.oid } (\text{upd } n) \rangle$
shows $\langle \text{nn_pregraph } (\text{upd_neuron } G \text{ upd}) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{nn_pregraph_upd}_{\varphi}[\text{simp}]$: $\langle \text{nn_pregraph } (\text{upd_neuron } G (\text{upd}_{\varphi} \varphi' n_{id})) \rangle$
and $\text{nn_pregraph_upd}_{\beta}[\text{simp}]$: $\langle \text{nn_pregraph } (\text{upd_neuron } G (\text{upd}_{\beta} \beta' n_{id})) \rangle$
and $\text{nn_pregraph_upd}_{\alpha}[\text{simp}]$: $\langle \text{nn_pregraph } (\text{upd_neuron } G (\text{upd}_{\alpha} \alpha' n_{id})) \rangle$
 $\langle \text{proof} \rangle$

end

4.3.4 Updating arcs (edges)

context nn_pregraph **begin**

lemma $\text{upd}_{\omega_tl_immutable}[\text{simp}]$: $\langle \text{tl } a = \text{tl } (\text{upd}_{\omega} \omega' n_{hd} n_{tl} a) \rangle$
and $\text{upd}_{\omega_hd_immutable}[\text{simp}]$: $\langle \text{hd } a = \text{hd } (\text{upd}_{\omega} \omega' n_{hd} n_{tl} a) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{upd}_{\omega_ends_immutable}[\text{simp}]$: $\langle \text{arc_to_ends } G \ a = \text{arc_to_ends } G \ (\text{upd}_{\omega} \omega' n_{hd} n_{tl} a) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{upd_edge_tail_immutable}$:
 $\langle \text{tail } (\text{upd_edge } G \ \text{upd}) = \text{tail } G \ \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{upd_edge_head_immutable}$:
 $\langle \text{head } (\text{upd_edge } G \ \text{upd}) = \text{head } G \ \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{upd_edge_vert_immutable}$: $\langle \text{verts } (\text{upd_edge } G \ \text{upd}) = \text{verts } G \rangle$
 $\langle \text{proof} \rangle$

lemma upd_edge_arcs : $\langle a \in \text{arcs } (\text{upd_edge } G \ \text{upd}) \implies \exists x \in \text{arcs } G. a = \text{upd } x \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{wf_digraph_update_edge}$:
assumes $\langle \forall a \in \text{arcs } G. (\text{arc_to_ends } G \ a = \text{arc_to_ends } G \ (\text{upd } a)) \rangle$
shows $\langle \text{wf_digraph } (\text{upd_edge } G \ \text{upd}) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{fin_digraph_update_edge}$:
assumes $\langle \forall a \in \text{arcs } G. (\text{arc_to_ends } G \ a = \text{arc_to_ends } G \ (\text{upd } a)) \rangle$
shows $\langle \text{fin_digraph } (\text{upd_edge } G \ \text{upd}) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{nomulti_digraph_update_edge}$:
assumes $\langle \forall a \in \text{arcs } G. (\text{arc_to_ends } G \ a = \text{arc_to_ends } G \ (\text{upd } a)) \rangle$
shows $\langle \text{nomulti_digraph } (\text{upd_edge } G \ \text{upd}) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{loopfree_digraph_update_edge}$:

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

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 $\langle \text{nn_pregraph } (\text{upd_edge } G \ \text{upd}) \rangle$
 $\langle \text{proof} \rangle$

lemma *nn_pregraph_upd $_{\omega}$ [simp]*: $\langle \text{nn_pregraph } (\text{upd_edge } G \ (\text{upd}_{\omega} \ \omega' \ n_{hd} \ n_{tl})) \rangle$
 $\langle \text{proof} \rangle$

end

record $\langle 'a, 'b, 'c \rangle$ *neural_network* =
graph :: $\langle ('a, 'b) \text{ neuron}, ('a, 'b) \text{ edge} \rangle$ *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**
 $\langle \text{upd_edge}' \ N \ \text{upd} = \langle \langle \langle \text{graph} = \text{upd_edge } (\text{graph } N) \ \text{upd}, \text{activation_tab} = \text{activation_tab } N \rangle \rangle \rangle$
 $\rangle \rangle$

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**
 $\langle \text{upd_neuron}' \ N \ \text{upd} = \langle \langle \langle \text{graph} = \text{upd_neuron } (\text{graph } N) \ \text{upd}, \text{activation_tab} = \text{activation_tab } N \rangle \rangle \rangle$
 $\rangle \rangle$

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 *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 *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 $\langle \text{sorted_list_of_set}' \equiv \text{map_fun } id \ id \ (\text{folding_on_F } (\text{insert_key } (\lambda \ x. \ \text{uid } (\text{tl } x)))) \ \square \rangle$

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

context $\text{nn_pregraph } \text{begin}$

lemma $\text{incoming_arcs_l_eq_incoming_arcs} : \langle \text{set } (\text{incoming_arcs_l } N \ n_{id}) = (\text{incoming_arcs } N \ n_{id}) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{incoming_arcs_l_alt_def} : \langle (\text{incoming_arcs_l } N \ n_{id})$
 $= (\text{sorted_key_list_of_set } (\lambda \ x. \ \text{uid } (\text{tl } x)) (\text{incoming_arcs } N \ n_{id})) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{insert_key_comm} : \text{inj } f \Longrightarrow (\text{insert_key } f \ y \circ \text{insert_key } f \ x) = (\text{insert_key } f \ x \circ \text{insert_key } f \ y)$
 $\langle \text{proof} \rangle$

lemma $\text{tl_subset_verts} : \langle \text{tl } ' (\text{arcs } G) \subseteq \text{verts } G \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{hd_subset_verts} : \langle \text{hd } ' (\text{arcs } G) \subseteq \text{verts } G \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{inj_on_tl} : \langle \text{inj_on } \text{uid } (\text{tl } ' (\text{arcs } G)) \rangle$
 $\langle \text{proof} \rangle$

end

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

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

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

locale $\text{nn_graph} = \text{nn_pregraph} +$
assumes $\text{id_vert_inj} : \langle \text{inj_on } \text{uid } (\text{verts } G) \rangle$
and $\text{inputs_In} :$
 $\langle \text{input_verts } G = \text{Set.filter } (\lambda \ v. \ (\text{case } v \ \text{of } \text{In } _ \Rightarrow \text{True} \ | \ _ \Rightarrow \text{False})) (\text{verts } G) \rangle$
and $\text{outputs_Out} :$
 $\langle \text{output_verts } G = \text{Set.filter } (\lambda \ v. \ (\text{case } v \ \text{of } \text{Out } _ \Rightarrow \text{True} \ | \ _ \Rightarrow \text{False})) (\text{verts } G) \rangle$
and $\text{internal_Neuron} :$
 $\langle \text{internal_verts } G = \text{Set.filter } (\lambda \ v. \ (\text{case } v \ \text{of } \text{Neuron } _ \Rightarrow \text{True} \ | \ _ \Rightarrow \text{False})) (\text{verts } G) \rangle$
begin

lemma $\text{nn_graph} : \text{nn_graph } G \ \langle \text{proof} \rangle$
end

locale $\text{neural_network_digraph} =$

fixes $N::\langle('a::\{\text{comm_monoid_add,times,linorder,one}\}, 'b, 'c) \text{neural_network}\rangle$
 assumes $\langle \text{nn_graph } (\text{graph } N) \rangle$
 and $\langle \varphi ' \{n . \text{Neuron } n \in (\text{verts } (\text{graph } N)) \} \subseteq \text{dom } (\text{activation_tab } N) \rangle$

4.3.5 The empty neural network

definition $\text{empty}::\langle('a, 'b) \text{nn_pregraph}\rangle$ where
 $\langle \text{empty} = (\{\text{verts}=\{\}, \text{arcs}=\{\}, \text{tail}=\text{edge.tl}, \text{head}=\text{edge.hd}\}) \rangle$

lemma $\text{nn_pregraph_empty}[\text{simp}]:\langle \text{nn_pregraph } (\text{empty}) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{nn_graph_empty}[\text{simp}]:\langle \text{nn_graph } (\text{empty}) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{fold_inv}: P e \implies (\forall e' x. P e' \longrightarrow P (f x e')) \implies P (\text{fold } f \text{ xs } e)$
 $\langle \text{proof} \rangle$

lemma $\text{nn_pregraph_fold}:\langle \text{nn_pregraph } G \implies \text{nn_pregraph } (\text{foldr } (\lambda a b. \text{add_nn_edge } b a) \text{ edge_list } G) \rangle$
 $\langle \text{proof} \rangle$

definition

$\langle \text{mk_nn_pregraph } \text{edge_list} = \text{foldr } (\lambda a b. \text{add_nn_edge } b a) \text{ edge_list } \text{empty} \rangle$

lemma $\text{nn_pregraph_mk}:\langle \text{nn_pregraph}(\text{mk_nn_pregraph } \text{edge_list}) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{verts_subsetq_add_edge}:\text{nn_pregraph } G \implies \text{verts } G \subseteq \text{verts } (\text{add_nn_edge } G a)$
 $\langle \text{proof} \rangle$

4.3.6 Computing Predictions of Neural Networks

datatype $\text{error} = \text{OK} \mid \text{ERROR}$

locale $\text{neural_network_digraph_single} = \text{neural_network_digraph } N$
 for $N::\langle('a::\{\text{comm_monoid_add,times,linorder,one}\}, 'b, 'a \Rightarrow 'a) \text{neural_network}\rangle$

function $(\text{sequential}, \text{domintros}) \text{predict}_{\text{digraph_single}} 'n::\text{nat}$
 $\Rightarrow ('a::\{\text{comm_monoid_add,times,linorder,one}\}, 'b, 'a \Rightarrow 'a) \text{neural_network}$
 $\Rightarrow (\text{id} \rightarrow 'a) \Rightarrow ('a, 'b) \text{edge} \Rightarrow ('a \times \text{error})$

where

$\langle \text{predict}_{\text{digraph_single}} 'n \text{ inputs } ((\omega=_, \text{tl}=_, \text{hd}=\text{In } _)) = (\text{o}, \text{ERROR}) \rangle$
 $\mid \langle \text{predict}_{\text{digraph_single}} 'n \text{ inputs } ((\omega=_, \text{tl}=\text{Out } _, \text{hd}=_)) = (\text{o}, \text{ERROR}) \rangle$
 $\mid \langle \text{predict}_{\text{digraph_single}} 'n \text{ inputs } ((\omega=\omega', \text{tl}=\text{In } \text{uid}_{in}, \text{hd}=_)) = (\text{case } \text{inputs } \text{uid}_{in} \text{ of}$
 $\quad \text{None} \Rightarrow (\text{o}, \text{ERROR})$
 $\quad \mid \text{Some } v \Rightarrow (v * \omega', \text{OK}) \rangle$
 $\mid \langle \text{predict}_{\text{digraph_single}} 'n \text{ inputs } e = (\text{if } \text{o} < n \text{ then}$
 $\quad (\text{let}$
 $\quad \quad \omega' = \omega e;$
 $\quad \quad \text{tl}' = (\text{case } (\text{tl } e) \text{ of } (\text{Neuron } t) \Rightarrow t);$
 $\quad \quad E' = \text{incoming_arcs } N (\text{Neuron.oid } \text{tl}');$
 $\quad \quad \text{lvals} = ((\lambda e'. (\text{case } \text{predict}_{\text{digraph_single}} '(n-1) \text{ inputs } e' \text{ of}$

```

      (⊥, 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) )>
⟨proof⟩

```

termination

⟨proof⟩

definition

```

⟨predictdigraph_single N inputs e = (case predictdigraph_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

```

⟨ predictdigraph_single_list N inputs' = those (map (λ e. predictdigraph_single N (mk_input_map N inputs') e)
  (get_output_edge_l N))⟩

```

context neural_network_digraph_single begin

lemma ids_growing': ⟨neural_network_digraph N ⇒ e ∈ edges N ⇒ uid (tl e) < uid (hd e)⟩
 ⟨proof⟩

end

context neural_network_digraph begin

fun (sequential) predict_{digraph}::⟨id → 'a⟩ list ⇒ ('a, 'b) edge list ⇒ ('a list × error)⟩

where

```

⟨predictdigraph _ _ = ([], ERROR)⟩

```

end

record 'a data =

inputs::id → 'a

outputs::id → 'a

end

4.4 Main Theory (Digraph) (≡ NN_Digraph_Main)

theory

NN_Digraph_Main

```
imports  
  NN_Common  
  NN_Digraph  
  Activation_Functions  
  Properties  
begin  
  
   $\langle ML \rangle$   
  
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 $finite_elements_mat: finite\ (elements_mat\ A)$
 $\langle proof \rangle$

lemma $max_{mat_is_element}:$
shows $\langle elements_mat\ m \neq \{\} \implies max_{mat}\ m \in elements_mat\ m \rangle$
 $\langle proof \rangle$

lemma $min_{mat_is_element}:$
 $\langle elements_mat\ m \neq \{\} \implies min_{mat}\ m \in elements_mat\ m \rangle$
 $\langle proof \rangle$

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 $\langle list_of_vec\ m \neq [] \implies max_{vec}\ m \in set\ (list_of_vec\ m) \rangle$
 $\langle proof \rangle$

lemma $\text{min}_{vec_is_element}$:

shows $\langle \text{list_of_vec } m \neq [] \implies \text{min}_{vec} m \in \text{set}(\text{list_of_vec } m) \rangle$

$\langle \text{proof} \rangle$

lemma $\text{max}_{vec_vCons_append_eq}$: $\langle \text{max}_{vec} (\text{vCons } x \text{ } xs) = \text{max}_{vec} xs \vee \text{max}_{vec} (\text{vCons } x \text{ } xs) = x \rangle$

$\langle \text{proof} \rangle$

lemma $\text{max}_{vec_append_eq}$: $\langle \text{max}_{vec} (\text{vec_of_list } (xs @ [x])) = \text{max}_{vec} (\text{vec_of_list } xs) \vee \text{max}_{vec} (\text{vec_of_list } (xs @ [x])) = x \rangle$

$\langle \text{proof} \rangle$

lemma $\text{min}_{vec_vCons_append_eq}$: $\langle \text{min}_{vec} (\text{vCons } x \text{ } xs) = \text{min}_{vec} xs \vee \text{min}_{vec} (\text{vCons } x \text{ } xs) = x \rangle$

$\langle \text{proof} \rangle$

lemma $\text{min}_{vec_append_eq}$: $\langle \text{min}_{vec} (\text{vec_of_list } (xs @ [x])) = \text{min}_{vec} (\text{vec_of_list } xs) \vee \text{min}_{vec} (\text{vec_of_list } (xs @ [x])) = x \rangle$

$\langle \text{proof} \rangle$

lemma $\text{max}_{vec_vec_cons_eq}$: $\langle \text{max}_{vec} ((\text{vec_of_list } [x]) @_v xs) = \text{max}_{vec} xs \vee \text{max}_{vec} ((\text{vec_of_list } [x]) @_v xs) = x \rangle$

$\langle \text{proof} \rangle$

lemma $\text{max}_{vec_cons_eq}$: $\langle \text{max}_{vec} (\text{vec_of_list } (x\#xs)) = \text{max}_{vec} (\text{vec_of_list } xs) \vee \text{max}_{vec} (\text{vec_of_list } (x\#xs)) = x \rangle$

$\langle \text{proof} \rangle$

lemma $\text{min}_{vec_vec_cons_eq}$: $\langle \text{min}_{vec} ((\text{vec_of_list } [x]) @_v xs) = \text{min}_{vec} xs \vee \text{min}_{vec} ((\text{vec_of_list } [x]) @_v xs) = x \rangle$

$\langle \text{proof} \rangle$

lemma $\text{min}_{list_cons_eq}$: $\langle \text{min}_{vec} (\text{vec_of_list } (x\#xs)) = \text{min}_{vec} (\text{vec_of_list } xs) \vee \text{min}_{vec} (\text{vec_of_list } (x\#xs)) = x \rangle$

$\langle \text{proof} \rangle$

lemma $\text{max}_{vec_vec_append_limit}$: **assumes** $\langle xs \neq \text{vNil} \rangle$ **shows** $\langle \text{max}_{vec} xs \leq \text{max}_{vec} (\text{vCons } x \text{ } xs) \rangle$

$\langle \text{proof} \rangle$

lemma $\text{max}_{vec_append_limit}$: **assumes** $\langle xs \neq [] \rangle$ **shows** $\langle \text{max}_{vec} (\text{vec_of_list } xs) \leq \text{max}_{vec} (\text{vec_of_list } (xs @ [x])) \rangle$

$\langle \text{proof} \rangle$

lemma $\text{min}_{vec_vec_append_limit}$: **assumes** $\langle xs \neq \text{vNil} \rangle$ **shows** $\langle \text{min}_{vec} xs \geq \text{min}_{vec} (\text{vCons } x \text{ } xs) \rangle$

$\langle \text{proof} \rangle$

lemma $\text{min}_{vec_append_limit}$: **assumes** $\langle xs \neq [] \rangle$ **shows** $\langle \text{min}_{vec} (\text{vec_of_list } xs) \geq \text{min}_{vec} (\text{vec_of_list } (xs @ [x])) \rangle$

$\langle \text{proof} \rangle$

lemma $\text{max}_{vec_vec_cons_limit}$: **assumes** $\langle xs \neq \text{vNil} \rangle$ **shows** $\langle \text{max}_{vec} xs \leq \text{max}_{vec} ((\text{vec_of_list } [x]) @_v xs) \rangle$

$\langle \text{proof} \rangle$

lemma $\text{max}_{vec_cons_limit}$: **assumes** $\langle xs \neq [] \rangle$ **shows** $\langle \text{max}_{vec} (\text{vec_of_list } xs) \leq \text{max}_{vec} (\text{vec_of_list } (x\#xs)) \rangle$

$\langle \text{proof} \rangle$

lemma $\text{min}_{vec_vec_cons_limit}$: **assumes** $\langle xs \neq \text{vNil} \rangle$ **shows** $\langle \text{min}_{vec} xs \geq \text{min}_{vec} ((\text{vec_of_list } [x]) @_v xs) \rangle$

$\langle \text{proof} \rangle$

lemma $\text{min}_{vec_cons_limit}$: **assumes** $\langle xs \neq [] \rangle$ **shows** $\langle \text{min}_{vec} (\text{vec_of_list } xs) \geq \text{min}_{vec} (\text{vec_of_list } (x\#xs)) \rangle$

$\langle \text{proof} \rangle$

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

lemma `prediction2percentage_id:`
assumes `<max_vec p = 100>`
shows `<prediction2percentage p = p>`
`<proof>`

Maximum Prediction **definition** `posmax_of :: <'a::linorder Matrix.vec => (nat × 'a) option> where`
`<posmax_of l = (let m = max_vec l in find (\ e. snd e = m) (enumerate o (list_of_vec l)))>`
definition `pos_of_max :: <'a::linorder Matrix.vec => nat option> where`
`<pos_of_max l = map_option fst (posmax_of l)>`

definition `posmax_of' :: <'a::linorder Matrix.vec => (nat × 'a) option> where`
`<posmax_of' l = (let l' = list_of_vec l in`
`(if l' = [] then None`
`else Some ((hd o rev o (sort_key snd) o (enumerate o)) l'))>`

definition `pos_of_max' :: <'a::linorder Matrix.vec => nat option> where`
`<pos_of_max' l = map_option fst (posmax_of' l)>`

lemma `find_append_eq: <find P (xs@[x]) = (if find P xs = None then find P [x] else find P xs)>`
`<proof>`

Distance of Maximum Prediction to Next Highest Prediction **definition** `δ_min :: real mat => real where`
`<δ_min m = (let m' = max_mat m; m'' = Max (elements_mat m - {m'})`
`in |m' - m''|)>`

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 `zip_vec :: 'a Matrix.vec => 'b Matrix.vec => ('a × 'b) Matrix.vec where`
`zip_vec A B ≡ Matrix.vec (dim_vec A) (\ i. ((A $ i), (B $ i)))`

fun `map_vec2 :: (<'a => 'b => 'c>) => 'a Matrix.vec => 'b Matrix.vec => 'c Matrix.vec >`
where
`map_vec2 f xs ys = map_vec (\ (x,y). f x y) (zip_vec xs ys)`

fun `checkget_result_mat where`
`<checkget_result_mat _ None None = (None, True)>`
`|<checkget_result_mat ε (Some xs) (Some ys) = (Some xs, fold (∧) (list_of_vec (map_vec2 (\ x y. x ≈[ε] y) xs ys))`
`True)>`
`|<checkget_result_mat _ r = (r, False)>`

definition `<check_result_mat r ε s = snd (checkget_result_mat ε r s)>`

notation `check_result_mat (((_)/ ≈[_]) ≈m _) [60, 60] 60)`

definition `ensure_testdata_range_mat :: <real \Rightarrow real Matrix.vec list \Rightarrow (real Matrix.vec \rightarrow real Matrix.vec) \Rightarrow real Matrix.vec list \Rightarrow bool>`

where

`<ensure_testdata_range_mat delta inputs P outputs
= foldl (\wedge) True
(map (λ e. (P (fst e)) $\approx_{[\text{delta}]}$ Some (snd e))
(zip inputs outputs))>`

notation `ensure_testdata_range_mat ((_) \models_m {(_)} (_) {(_)} [61, 3, 90, 3] 60)`

Interval Arithmetic **definition** `<intervals_of_mat δ A = Matrix.vec (dim_vec A) (λ i. Interval((Ai) $-|\delta|$, (Ai) $+|\delta|$)) >`

definition `<intervals_of_m δ = map (intervals_of_mat δ) >`

fun `check_result_mat_interval_mat :: <'a::preorder Matrix.vec option \Rightarrow 'a interval Matrix.vec option \Rightarrow bool> where
<check_result_mat_interval_mat None None = True>
| <check_result_mat_interval_mat (Some xs) (Some ys) = fold (\wedge) (list_of_vec (map_vec2 (λ x y. x \in set_of y) xs ys)) True>
| <check_result_mat_interval_mat _ _ = False>`

notation `check_result_mat_interval_mat (((_) / \approx_m (_)) [60, 60] 60)`

We define `check_result_mat_interval` for checking that two matrices 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_mat :: <real Matrix.vec list \Rightarrow (real Matrix.vec \rightarrow real Matrix.vec) \Rightarrow real interval Matrix.vec list \Rightarrow bool>`

where

`<ensure_testdata_interval_mat inputs P outputs
= foldl (\wedge) True
(map (λ e . let a = (P (fst e)) in let b = Some (snd e) in (a \approx_m b))
(zip inputs outputs)) >`

notation `ensure_testdata_interval_mat (\models_{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 \Rightarrow (real Matrix.vec \rightarrow real Matrix.vec) \Rightarrow real Matrix.vec list \Rightarrow bool>`

where

`<ensure_testdata_max_mat inputs P outputs
= foldl (\wedge) True
(map (λ 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_mat (\models_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 =  $\langle$ ('b) ActivationRecord $\rangle$  +  
   $\beta$  ::  $\langle$ 'a $\rangle$   
   $\omega$  ::  $\langle$ 'c $\rangle$ 
```

```
datatype ('a, 'b, 'c) layer = In  $\langle$ InOutRecord $\rangle$   
  | Out  $\langle$ InOutRecord $\rangle$   
  | Dense  $\langle$ ('a, 'b, 'c) LayerRecord $\rangle$   
  | Activation  $\langle$ ('b) ActivationRecord $\rangle$ 
```

fun *units_l* **where**

```
 $\langle$ unitsl (In l) = units l  
|  $\langle$ unitsl (Out l) = units l  
|  $\langle$ unitsl (Dense l) = units l  
|  $\langle$ 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**

```
 $\langle$ isIn (In _) = True $\rangle$   
|  $\langle$ isIn _ = False $\rangle$ 
```

fun *isOut* **where**

```

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

```

fun isInternal where

```

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

```

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

record ('a, 'b, 'c) *neural_network_seq_layers* =
layers :: $\langle ('a, 'b, 'c) \text{ layer list} \rangle$
activation_tab :: $\langle 'b \Rightarrow (('a \Rightarrow 'a) \text{ option}) \rangle$

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>
```

```
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*: $1\text{-lipschitz_on } (X::\text{real set}) \text{ (relu)}$

```
<proof>
```

lemma *identity_lipschitz*: $1\text{-lipschitz_on } (X::\text{real set}) \text{ (identity)}$

```
<proof>
```

Neural Network: Layers lemma *input_output_lipschitz_continuous*:

```
< $1\text{-lipschitz\_on } (U::\text{real set}) \text{ } (\lambda i . i)$ >
```

```
<proof>
```

lemma *activation_lipschitz_continuous*:

```
  assumes < $C\text{-lipschitz\_on } U \text{ } f$ >
```

```
  shows < $C\text{-lipschitz\_on } U \text{ } (\lambda i . f i)$ >
```

```
<proof>
```

lemma *lipschitz_on_add_const*:

```
  shows < $(1::\text{real})\text{-lipschitz\_on } (U::\text{real set}) \text{ } (\lambda x . x + c)$ >
```

```
<proof>
```

lemma *lipschitz_on_fold_add*:

```
  shows  $1\text{-lipschitz\_on } (U::\text{real set}) \text{ (fold (+) xs)$ 
```

```
<proof>
```

lemma *lipschitz_on_fold_add_zero*:

```
  shows  $1\text{-lipschitz\_on } (U::\text{real set}) \text{ } (\lambda x . \text{fold (+) [x] } (o::\text{real}))$ 
```

```
<proof>
```

lemma *lipschitz_on_foldr_add*:

```
  shows  $1\text{-lipschitz\_on } (U::\text{real set}) \text{ } (\lambda s . \text{foldr (+) xs } s)$ 
```

```
<proof>
```

lemma *lipschitz_on_sumlist_rev*:
shows $1\text{-lipschitz_on } (U::\text{real set}) ((+) (\text{sum_list } (\text{rev } xs)))$
<proof>

lemma *lipschitz_on_sumlist*:
shows $1\text{-lipschitz_on } (U::\text{real set}) ((+) (\text{sum_list } xs))$
<proof>

lemma *lipschitz_on_mult_const*:
shows $|c|\text{-lipschitz_on } (U::\text{real set}) (\lambda x . x * c)$
<proof>

lemma *lipschitz_on_weighted_sum_single*:
 $|w|\text{-lipschitz_on } (U::\text{real set}) (\lambda x . x * w + b)$
<proof>

lemma *lipschitz_on_fold_add_zero'*:
shows $2\text{-lipschitz_on } (U::\text{real set}) (\lambda x . (\text{fold } (+) [x,x] (o::\text{real})) + w)$
<proof>

lemma *lipschitz_on_mult_const'*:
shows $\langle \forall x \in \text{set } xs . |c|\text{-lipschitz_on } (\text{set } xs) (\lambda y . c * y) \rangle$
<proof>

typedef (*'a*, *'nr::finite*, *'nc::finite*) *fixed_mat* =
carrier_mat (*CARD*(*'nr*)) (*CARD*(*'nc*)) :: *'a mat set*
morphisms *Rep_fixed_mat Abs_fixed_mat* *<proof>*

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*
<proof>

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* (*o_m* (*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*, *'n::finite*) *fixed_vec* **where**

$fixed_vec_zero = Abs_fixed_vec (o_v (CARD('nr)))$

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 $\lambda c A. c \cdot_m A$
 $\langle proof \rangle$

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

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

lift_definition $fixed_vec_smult :: 'a::times \Rightarrow ('a, 'nr::finite) fixed_vec \Rightarrow ('a, 'nr) fixed_vec$
is $\lambda c A. c \cdot_v A$
 $\langle proof \rangle$

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 $\lambda v A. vec (dim_col A) (\lambda i. col A i \cdot v)$
 $\langle proof \rangle$

lift_definition $map_fixed_vec :: ('a \Rightarrow 'b) \Rightarrow ('a, 'nr::finite) fixed_vec \Rightarrow ('b, 'nr::finite) fixed_vec$
is $map_vec :: ('a \Rightarrow 'b) \Rightarrow 'a\ vec \Rightarrow 'b\ vec$
 $\langle proof \rangle$

lemma $zero_in_carrier:$
 $o_m (CARD('nr)) (CARD('nc)) \in carrier_mat (CARD('nr)) (CARD('nc))$
 $\langle proof \rangle$

lemma $Rep_fixed_mat_zero [simp]:$
 $Rep_fixed_mat (fixed_mat_zero :: ('a::zero, 'nr::finite, 'nc::finite) fixed_mat) = o_m (CARD('nr)) (CARD('nc))$
 $\langle proof \rangle$

lemma $Rep_fixed_mat_add [simp]:$
 $Rep_fixed_mat (fixed_mat_add A B) = Rep_fixed_mat A + Rep_fixed_mat B$
 $\langle proof \rangle$

lemma $Rep_fixed_vec_zero [simp]:$
 $Rep_fixed_vec (fixed_vec_zero :: ('a::zero, 'n::finite) fixed_vec) = o_v (CARD('n))$
 $\langle proof \rangle$

lemma $Rep_fixed_vec_add [simp]:$
 $Rep_fixed_vec (fixed_vec_add A B) = Rep_fixed_vec A + Rep_fixed_vec B$
 $\langle proof \rangle$

lemma $Rep_fixed_mat_inject: Rep_fixed_mat A = Rep_fixed_mat B \implies A = B$
 $\langle proof \rangle$

lemma $Rep_fixed_vec_inject: Rep_fixed_vec A = Rep_fixed_vec B \implies A = B$
 $\langle proof \rangle$

lift_definition *row_fixed* :: ('a, 'n::finite, 'm::finite) fixed_mat \Rightarrow nat \Rightarrow ('a, 'm) fixed_vec is
 $\lambda A i. \text{vec } (\text{CARD}('m)) (\lambda j. A \$\$ (i, j))$
<proof>

lift_definition *col_fixed* :: ('a, 'n::finite, 'm::finite) fixed_mat \Rightarrow nat \Rightarrow ('a, 'n) fixed_vec is
 $\lambda A j. \text{vec } (\text{CARD}('n)) (\lambda i. A \$\$ (i, j))$
<proof>

lemma $\text{CARD}(285) = 285$ *<proof>*

instantiation *fixed_mat* :: (semiring_1, finite, finite) times
begin

lift_definition *mat_mult* :: ('a::semiring_1, 'n::finite, 'm::finite) fixed_mat \Rightarrow
('a, 'm, 'k::finite) fixed_mat \Rightarrow
('a, 'n, 'k) fixed_mat is
 $\lambda A B. \text{mat } (\text{CARD}('n)) (\text{CARD}('k)) (\lambda (i, j).$
 $\text{sum_list } (\text{map } (\lambda l. A \$\$ (i, l) * B \$\$ (l, j)) [0..<\text{CARD}('m)]))$
<proof>

instance *<proof>*

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 \Rightarrow ('a, 'nr, 'nc) fixed_mat \Rightarrow ('a, 'nr, 'nc) fixed_mat
where
plus_fixed_mat = *fixed_mat_add*

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

definition *scaleR_fixed_mat* :: real \Rightarrow ('a, 'nr::finite, 'nc::finite) fixed_mat \Rightarrow ('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 \Rightarrow real **where**
norm_fixed_mat A = $\text{sqrt } (\sum i \in \{0..<\text{CARD}('nr)\}. \sum j \in \{0..<\text{CARD}('nc)\}. (\text{norm } (\text{fixed_mat_index } A \ i \ j))^2)$

definition *dist_fixed_mat* :: ('a, 'nr::finite, 'nc::finite) fixed_mat \Rightarrow ('a, 'nr, 'nc) fixed_mat \Rightarrow 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 \times ('a, 'nr, 'nc) fixed_mat) filter **where**
uniformity_fixed_mat = (*INF* e $\in \{0 <.. \}$. *principal* {(x, y). *dist* x y < e})

definition *open_fixed_mat* :: ('a, 'nr::finite, 'nc::finite) fixed_mat set \Rightarrow bool **where**
open_fixed_mat S = $(\forall x \in S. \forall_F (x', y) \text{ in uniformity. } x' = x \longrightarrow y \in S)$

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

lemma *uminus_add*: $-(A :: ('a, 'nr::finite, 'nc::finite) \text{ fixed_mat}) + A = o$
<proof>

lemma *smult*: $a *_R b *_R x = (a * b) *_R (x :: ('a, 'nr::finite, 'nc::finite) \text{ fixed_mat})$
<proof>

lemma *scaleR*: $1 *_R x = (x :: ('a, 'nr::finite, 'nc::finite) \text{ fixed_mat})$
<proof>

lemma *scaleR_o*: $o *_R x = (o :: ('a, 'nr::finite, 'nc::finite) \text{ fixed_mat})$
<proof>

lemma *norm_o*: $\text{norm } (o :: ('a, 'nr::finite, 'nc::finite) \text{ fixed_mat}) = o$
<proof>

lemma *sgn*: $\text{sgn } x = \text{inverse } (\text{norm } x) *_R (x :: ('a, 'nr::finite, 'nc::finite) \text{ fixed_mat})$
<proof>

lemma *norm_eq_zero_iff*: $(\text{norm } x = (o :: \text{real})) = (x = (o :: ('a, 'nr::finite, 'nc::finite) \text{ fixed_mat}))$
<proof>

lemma *sum_tuple*: $\langle (\sum i < n. \sum j < m . P \ i \ j) = (\sum p \in \{(i,j). i < n \wedge j < m\}. P \ (\text{fst } p) \ (\text{snd } p)) \rangle$
<proof>

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

lemma *norm_scaleR*: $\text{norm } (a *_R x) = |a| * \text{norm } (x :: ('a, 'nr::finite, 'nc::finite) \text{ fixed_mat})$
<proof>

instance
<proof>
end

instantiation *fixed_vec* :: $(\{\text{real_normed_vector}, \text{times}, \text{one}, \text{real_algebra_1}\}, \text{finite}) \text{ real_normed_vector}$
begin

lift_definition *zero_fixed_vec* :: ('a, 'b) fixed_vec **is**
zero_vec (CARD('b))

<proof>

lift_definition *plus_fixed_vec* :: ('a, 'b) fixed_vec \Rightarrow ('a, 'b) fixed_vec \Rightarrow ('a, 'b) fixed_vec is
fixed_vec_add *<proof>*

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
 $\lambda v. \text{smult_vec } (-1) v$
<proof>

lift_definition *minus_fixed_vec* :: ('a, 'b) fixed_vec \Rightarrow ('a, 'b) fixed_vec \Rightarrow ('a, 'b) fixed_vec is
 $\lambda v w. v + (\text{smult_vec } (-1) w)$
<proof>

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

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 \times ('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*: $-(A :: ('a, 'n::finite) \text{fixed_vec}) + A = 0$
<proof>

lemma *smult_vec*: $a *_R b *_R x = (a * b) *_R (x :: ('a, 'n::finite) \text{fixed_vec})$
<proof>

lemma *scaleR_vec*: $1 *_R x = (x :: ('a, 'n::finite) \text{fixed_vec})$
<proof>

lemma *norm_o_vec*: $\text{norm } (0 :: ('a, 'n::finite) \text{fixed_vec}) = 0$
<proof>

lemma *scaleR_o_vec*: $0 *_R x = (0 :: ('a, 'n::finite) \text{fixed_vec})$
<proof>

lemma *sgn_vec*: $\text{sgn } x = \text{inverse } (\text{norm } x) *_R (x :: ('a, 'n::finite) \text{fixed_vec})$
<proof>

lemma norm_eq_zero_iff_vec: $(\text{norm } x = (0::\text{real})) = (x = (0::('a, 'n::\text{finite}) \text{fixed_vec}))$
<proof>

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

lemma norm_scaleR_vec: $\text{norm } (a *_R x) = |a| * \text{norm } (x::('a, 'n::\text{finite}) \text{fixed_vec})$
<proof>

instance
<proof>

end

lemma uminus_fixed_vec:
assumes $(v::'a::\{\text{real_algebra_1, real_normed_vector}\} \text{Matrix.vec}) \in \text{carrier_vec } (\text{CARD}('n::\text{finite}))$
shows $-\text{Abs_fixed_vec } v = (\text{Abs_fixed_vec } (-v))::('a::\{\text{real_algebra_1, real_normed_vector}\}, 'n::\text{finite}) \text{fixed_vec}$
<proof>

lemma lipschitz_on_mat_add:
shows $\langle (1::\text{real})-\text{lipschitz_on } U (\lambda (A::('a::\{\text{real_algebra_1, real_normed_vector}\}, 'nr::\text{finite}, 'nc::\text{finite}) \text{fixed_mat}) . A + M) \rangle$
<proof>

lemma vec_minus_element:
fixes $v w :: 'a::\{\text{minus, zero}\} \text{vec}$
assumes $\text{dim_vec } v = \text{dim_vec } w$ **and** $i < \text{dim_vec } v$
shows $\text{vec_index } (v - w) i = \text{vec_index } v i - \text{vec_index } w i$
<proof>

lemma vec_minus:
fixes $v w :: 'a::\{\text{minus, zero}\} \text{vec}$
assumes $\text{dim_vec } v = \text{dim_vec } w$ **and** $i < \text{dim_vec } v$
shows $(v - w) = \text{vec } (\text{dim_vec } v) (\lambda i. \text{vec_index } v i - \text{vec_index } w i)$
<proof>

lemma Rep_fixed_vec_plus:
 $\text{Rep_fixed_vec } ((u::('a::\{\text{real_algebra_1, real_normed_vector}\}, 'n::\text{finite}) \text{fixed_vec}) + (v::('a::\{\text{real_algebra_1, real_normed_vector}\}, 'n::\text{finite}) \text{fixed_vec})) = \text{Rep_fixed_vec } u + \text{Rep_fixed_vec } v$
<proof>

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>

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)

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 $i < \text{dim_col } (A::'a::\{\text{semiring_o, ab_semigroup_mult}\} \text{ Matrix.mat})$
shows $(v \cdot v * A) \$ i = (\sum j = 0..<\text{dim_row } A. v \$ j * A \$\$ (j, i))$
(proof)

lemma Rep_fixed_mat_minus:

$\text{Rep_fixed_mat } ((x - y)::('a, 'b, 'c) \text{ fixed_mat}) = \text{Rep_fixed_mat } x - \text{Rep_fixed_mat } (y::('a::\{\text{real_algebra_1, real_normed_vector}\}, 'b::\text{finite}, 'c::\text{finite}) \text{ fixed_mat})$
(proof)

lemma vector_matrix_inequality:

fixes $c :: ('a::\{\text{real_normed_field, real_inner}\}, 'nr::\text{finite}) \text{ fixed_vec}$
and $U :: ('a, 'nr, 'nc::\text{finite}) \text{ fixed_mat set}$
and $C :: \text{real}$
assumes $C_bound: C \geq \text{norm } c$
and $C_nonneg: C \geq 0$
shows $\bigwedge x y. x \in U \implies y \in U \implies$
 $\text{sqrt } (\sum i = 0..<\text{CARD } ('nc). (\text{norm } (\text{fixed_vec_index } (c \cdot_{fv} * x - c \cdot_{fv} * y) i))^2) \leq$
 $C * \text{sqrt } (\sum i = 0..<\text{CARD } ('nr). \sum j = 0..<\text{CARD } ('nc). (\text{norm } (\text{fixed_mat_index } (x - y) i j))^2)$
(proof)

lemma lipschitz_on_mat_mult:

assumes $\langle 0 \leq C \rangle$ **and** $\text{norm } c \leq C$
shows $\langle C - \text{lipschitz_on } U (\lambda (y::('a::\{\text{real_inner, real_normed_field}\}, 'nr::\text{finite}, 'nc::\text{finite}) \text{ fixed_mat}).$
 $(c::('a, 'nr) \text{ fixed_vec}) \cdot_{fv} * y) \rangle$
(proof)

lemma lipschitz_on_weighted_sum_bias:

assumes $\langle 0 \leq C \rangle$ **and** $\text{norm } c \leq C$
shows $\langle C - \text{lipschitz_on } U (\lambda (y::('a::\{\text{real_inner, real_normed_field}\}, 'nr::\text{finite}, 'nc::\text{finite}) \text{ fixed_mat}). (c \cdot_{fv} * y) +$
 $b) \rangle$
(proof)

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 $(v1 - v2) \cdot v * A = v1 \cdot v * A - v2 \cdot v * (A::'a::\{\text{real_normed_field, real_inner}\} \text{ mat})$
(proof)

lemma matrix_vector_inequality:

fixes $c :: ('a::\{\text{real_normed_field, real_inner}\}, 'nr::\text{finite}, 'nc::\text{finite}) \text{ fixed_mat}$
and $U :: ('a, 'nr) \text{ fixed_vec set}$
and $C :: \text{real}$
assumes $C_bound: C \geq \text{norm } c$
and $C_nonneg: C \geq 0$
shows $\bigwedge x y. x \in U \implies y \in U \implies$
 $\text{sqrt } (\sum i = 0..<\text{CARD } ('nc). (\text{norm } (\text{fixed_vec_index } (x \cdot_{fv} * c - y \cdot_{fv} * c) i))^2) \leq$

$C * \text{sqrt} (\sum_{i=0..<CARD('nr)}. (\text{norm} (\text{fixed_vec_index} (x - y) i))^2)$
 <proof>

lemma lipschitz_on_vec_mult:

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

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

<proof>

lemma C_ge_norm:

$\text{norm} (c :: ('a :: \{\text{real_algebra}_1, \text{real_normed_vector}\}, 'nr :: \text{finite}, 'nc :: \text{finite}) \text{fixed_mat}) \leq C \implies 0 \leq C$

<proof>

lemma lipschitz_on_weighted_sum_bias':

assumes $\text{norm } c \leq C$

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

<proof>

lemma lipschitz_on_dense:

assumes $\text{norm } c \leq C$

assumes $D\text{-lipschitz_on } ((\lambda y . (c :: ('a :: \{\text{real_inner}, \text{real_normed_field}\}, 'n :: \text{finite}) \text{fixed_vec})_{fv} * y + b) 'U) f$

shows $(D * C)\text{-lipschitz_on } U (\lambda y . f ((c_{fv} * y) + b))$

<proof>

lemma lipschitz_on_dense':

assumes $\text{norm } c \leq C$

assumes $D\text{-lipschitz_on } ((\lambda y . y_{fv} * (c :: ('a :: \{\text{real_inner}, \text{real_normed_field}\}, 'nr :: \text{finite}, 'nc :: \text{finite}) \text{fixed_mat}) + b) 'U) f$

shows $(D * C)\text{-lipschitz_on } U (\lambda y . f ((y_{fv} * c) + b))$

<proof>

lemma lipschitz_on_input_output:

shows $1\text{-lipschitz_on } U (\lambda i . i)$

<proof>

lemma lipschitz_on_activation:

assumes $C\text{-lipschitz_on } U f$

shows $C\text{-lipschitz_on } U (\lambda i . f i)$

<proof>

Neural Network: Layers **fun** $\text{predict}_{\text{layer}_f} :: ((('a :: \{\text{real_algebra}_1, \text{real_normed_vector}\}, 'b :: \text{finite}) \text{fixed_vec}, 'c, 'd) \text{neural_network_seq_layers}$

$\implies ('a, 'b :: \text{finite}) \text{fixed_vec} \implies (('a, 'b) \text{fixed_vec}, 'c, ('a, 'b, 'b) \text{fixed_mat}) \text{layer} \implies ('a, 'b) \text{fixed_vec}$

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*:

$1\text{-lipschitz_on } U ((\lambda \text{ vs} . (\text{predict}_{\text{layer}_f} N \text{ vs } (\text{In } x1))))$
 $\langle \text{proof} \rangle$

lemma *output_layer_lipschitz_on*:

$1\text{-lipschitz_on } U ((\lambda \text{ vs} . (\text{predict}_{\text{layer}_f} N \text{ vs } (\text{Out } x1))))$
 $\langle \text{proof} \rangle$

lemma *dense_layer_lipschitz_on*:

assumes $\text{norm } (\omega x1) \leq C$
assumes $D\text{-lipschitz_on } ((\lambda y. y_{fv} * \omega x1 + \beta x1) 'U) (\text{the } (\text{activation_tab } N (\varphi x1)))$
shows $(C * D)\text{-lipschitz_on } U (\lambda \text{ vs} :: ('a :: \{\text{real_inner, real_normed_field}\}, 'c :: \text{finite}) \text{ fixed_vec} . (\text{predict}_{\text{layer}_f} N \text{ vs } (\text{Dense } x1)))$
 $\langle \text{proof} \rangle$

lemma *activation_layer_lipschitz_on*:

assumes $C\text{-lipschitz_on } U (\text{the } (\text{activation_tab } N (\varphi x1)))$
shows $C\text{-lipschitz_on } U (\lambda \text{ vs} . (\text{predict}_{\text{layer}_f} N \text{ vs } (\text{Activation } x1)))$
 $\langle \text{proof} \rangle$

lemma *foldl_layer_lipschitz_on*:

fixes $N :: (('a :: \{\text{real_algebra}_1, \text{real_normed_vector}\}, 'b :: \text{finite}) \text{ fixed_vec}, 'c, 'd) \text{ neural_network_seq_layers}$
assumes $\text{layer_lipschitz}: \forall l \in \text{set } ls. \exists C. C\text{-lipschitz_on } U (\lambda \text{ vs}. \text{predict}_{\text{layer}_f} N \text{ vs } l)$
assumes $\text{domain_invariant}: \forall l \in \text{set } ls. \forall \text{ vs} \in U. \text{predict}_{\text{layer}_f} N \text{ vs } l \in U$
shows $\exists C. C\text{-lipschitz_on } U (\lambda \text{ vs}. \text{foldl } (\text{predict}_{\text{layer}_f} N) \text{ vs } ls)$
 $\langle \text{proof} \rangle$

lemma *layers_lipschitz_from_components*:

fixes $N :: (('a :: \{\text{real_algebra}_1, \text{real_normed_vector, real_inner, real_normed_field}\}, 'b :: \text{finite}) \text{ fixed_vec}, 'c, 'd) \text{ neural_network_seq_layers}$
assumes $\text{weight_bounded}: \forall l \in \text{set } ls. (\text{case } l \text{ of Dense } pl \Rightarrow \text{norm } (\omega pl) \leq W)$
assumes $\text{activation_lipschitz}: \forall l \in \text{set } ls. (\text{case } l \text{ of}$
 $\text{Dense } pl \Rightarrow \exists D. D\text{-lipschitz_on } ((\lambda y. y_{fv} * \omega pl + \beta pl) 'U) (\text{the } (\text{activation_tab } N (\varphi pl))) \mid$
 $\text{Activation } pl \Rightarrow \exists C. C\text{-lipschitz_on } U (\text{the } (\text{activation_tab } N (\varphi pl))))$
shows $\forall l \in \text{set } ls. \exists C. C\text{-lipschitz_on } U (\lambda \text{ vs}. \text{predict}_{\text{layer}_f} N \text{ vs } l)$
 $\langle \text{proof} \rangle$

lemma *Rep_fixed_vec_minus*: $\text{Rep_fixed_vec } (x - y) = \text{Rep_fixed_vec } x - \text{Rep_fixed_vec } y$

$\langle \text{proof} \rangle$

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⟩

lemma *relu_lipschitz_fv*: 1-lipschitz_on (X::(real, 'b::finite) fixed_vec set) (map_fixed_vec relu)
⟨proof⟩

lemma *identity_lipschitz_fv*: 1-lipschitz_on (X) (map_fixed_vec identity)
⟨proof⟩

lemma *softplus_lipschitz'*: $\bigwedge x y. (x::(real, 'b::finite) fixed_vec) \in X \implies$
 $(y::(real, 'b::finite) fixed_vec) \in X \implies$
 $dist ((map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (\lambda f v. Matrix.vec (dim_vec v) (\lambda i. f (v \$ i)))) softplus$
 $x)::(real, 'b) fixed_vec)$
 $(map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (\lambda f v. Matrix.vec (dim_vec v) (\lambda i. f (v \$ i)))) softplus y)$
 $\leq 1 * dist x y$
⟨proof⟩

lemma *softplus_lipschitz*: 1-lipschitz_on (X::(real, 'b::finite) fixed_vec set) (map_fixed_vec softplus)
⟨proof⟩

end

5.2 Models

5.2.1 Digraphs as Layers (≡ 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
⟨*layer_equiv* f g = ($\exists p p'. \forall xs. f xs = permute_list p' (f (permute_list p xs))$)⟩

lemma *mset_eq_layer_equiv*:

assumes ⟨mset xs = mset ys⟩

and ⟨mset (f xs) = mset (g ys)⟩

shows ⟨f \equiv_l g⟩

⟨proof⟩

fun *output_neuron* **where**

⟨*output_neuron* (In nid) = False⟩

| ⟨*output_neuron* (Out nid) = True⟩

| ⟨*output_neuron* (Neuron n) = False⟩

fun *input_neuron* **where**

⟨*input_neuron* (In nid) = True⟩

| ⟨*input_neuron* (Out nid) = False⟩

| $\langle \text{input_neuron } (\text{Neuron } n) = \text{False} \rangle$

fun *hidden_neuron* **where**

| $\langle \text{hidden_neuron } (\text{In } \text{id}) = \text{False} \rangle$
 | $\langle \text{hidden_neuron } (\text{Out } \text{id}) = \text{False} \rangle$
 | $\langle \text{hidden_neuron } (\text{Neuron } n) = \text{True} \rangle$

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

where

$\langle \text{mk_edge } N \omega' \varphi' \alpha' \beta' \text{id}' \text{id}' = (\omega = \omega',$
 $\text{tl} = (\text{the_elem } \{n . n \in \text{neurons } N \wedge \text{uid } n = \text{id}'\}),$
 $\text{hd} = \text{Neuron } (\varphi = \varphi', \alpha = \alpha', \beta = \beta', \text{uid} = \text{id}') \rangle$

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

where

$\langle \text{mk_out_edge } N \text{id}' \text{id}' = (\omega = 1,$
 $\text{tl} = (\text{the_elem } \{n . n \in \text{neurons } N \wedge \text{uid } n = \text{id}'\}),$
 $\text{hd} = \text{Out } \text{id}' \rangle$

definition *mk_new_ids* :: $\langle ('a::\{\text{one}\}, 'b, 'c) \text{neural_network} \Rightarrow \text{nat list} \rangle$

where

$\langle \text{mk_new_ids } N = \text{upt } (\text{Max}(\text{uids } (\text{graph } N)) + 1)$
 $(\text{Max}(\text{uids } (\text{graph } N)) + \text{card } (\text{output_layer_ids } N) + 1) \rangle$

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*: $\langle \text{length}(\text{mk_new_ids } N) = \text{length}(\text{sorted_list_of_set}(\text{output_layer_ids } N)) \rangle$

$\langle \text{proof} \rangle$

lemma *new_id_len_card*: $\langle \text{length}(\text{mk_new_ids } N) = \text{card}(\text{output_layer_ids } N) \rangle$

$\langle \text{proof} \rangle$

lemma *new_id_distinct*: $\langle \text{distinct}(\text{mk_new_ids } N) \rangle$

$\langle \text{proof} \rangle$

lemma *new_id_greater*:

assumes $\langle \text{card } (\text{output_layer_ids } N) > 0 \rangle$
shows $\langle \text{Min}(\text{set}(\text{mk_new_ids } N)) > \text{Max}(\text{uids } (\text{graph } N)) \rangle$
 $\langle \text{proof} \rangle$

lemma *new_id_sorted*:

shows $\langle \text{sorted } (\text{mk_new_ids } N) \rangle$
 $\langle \text{proof} \rangle$

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) ∩ set(mk_new_ids N) = {}
⟨proof⟩
```

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 ∀ x ∈ set(mk_new_ids N). x ∉ uids (graph N)
⟨proof⟩
```

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 layers_{digraph}::⟨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 o < 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 `layersdigraph` 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 $isDense_s :: \langle ('a, 'b) \text{ edge set} \Rightarrow \text{bool} \rangle$ **where**
 $\langle isDense_s e = ((\forall n' \in tl' e . \forall n'' \in hd' e . \exists e' \in e . tl e' = n' \wedge hd e' = n'')) \rangle$

definition $isActivation_s :: \langle ('a, 'b) \text{ edge set} \Rightarrow \text{bool} \rangle$ **where**
 $\langle isActivation_s e = ((\forall n' \in tl' e . \exists! e' \in e . tl e' = n') \wedge (\forall n'' \in hd' e . \exists! e'' \in e . hd e'' = n'')) \rangle$

definition $isInput_s :: \langle ('a, 'b) \text{ edge set} \Rightarrow \text{bool} \rangle$ **where**
 $\langle isInput_s e = (isDense_s e \wedge (\forall n \in hd' e . input_neuron n)) \rangle$

definition $isOutput_s :: \langle ('a, 'b) \text{ edge set} \Rightarrow \text{bool} \rangle$ **where**
 $\langle isOutput_s e = (isActivation_s e \wedge (\forall n''' \in hd' e . output_neuron n''')) \rangle$

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 $isDense_l :: \langle ('a, 'b) \text{ edge list} \Rightarrow \text{bool} \rangle$ **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 $isInput_l :: \langle ('a, 'b) \text{ edge list} \Rightarrow \text{bool} \rangle$ **where**
 $\langle isInput_l e = (isDense_l e \wedge \text{foldr } (\wedge) (\text{map } input_neuron (\text{map } hd e)) \text{ True}) \rangle$

definition $isActivation_l :: \langle ('a, 'b) \text{ edge list} \Rightarrow \text{bool} \rangle$ **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 $isOutput_l :: \langle ('a, 'b) \text{ edge list} \Rightarrow \text{bool} \rangle$ **where**
 $\langle isOutput_l e = (isActivation_l e \wedge \text{foldr } (\wedge) (\text{map } (output_neuron \circ hd) e) \text{ True}) \rangle$

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 $\langle \text{foldr } (\wedge) (\text{map } (output_neuron \circ hd) e) \text{ True} = (\forall n' \in hd' \text{ set } e . output_neuron n') \rangle$
 $\langle \text{proof} \rangle$

lemma $allInput$:

shows $\langle \text{foldr } (\wedge) (\text{map } (input_neuron \circ hd) e) \text{ True} = (\forall n' \in hd' \text{ set } e . input_neuron n') \rangle$
 $\langle \text{proof} \rangle$

lemma $forAll$:

$\langle (\forall n' \in \text{set } (\text{map } tl e) . \forall n'' \in \text{set } (\text{map } hd e) . \text{filter } (\lambda e' . tl e' = n' \wedge hd e' = n'') e \neq []) = (\forall n' \in tl' \text{ set } e . \forall n'' \in hd' \text{ set } e . \exists e' \in \text{set } e . tl e' = n' \wedge hd e' = n'') \rangle$
 $\langle \text{proof} \rangle$

lemma $isDense_l_isDense_s_equivalence$: $\langle isDense_l E = isDense_s (\text{set } E) \rangle$
 $\langle \text{proof} \rangle$

lemma $isInput_l_isInput_s_equivalence$: $\langle isInput_l E = isInput_s (\text{set } E) \rangle$

⟨proof⟩

lemma *isActivation_l_isActivation_s_equivalence*:
 assumes *distinct*: ⟨distinct E⟩
 shows ⟨isActivation_l E = isActivation_s (set E)⟩
 ⟨proof⟩

lemma *isOutput_l_isOutput_s_equivalence*:
 assumes *distinct*: ⟨distinct E⟩
 shows ⟨isOutput_l E = isOutput_s (set E)⟩
 ⟨proof⟩

We currently support the following 4 types of layers:

definition ⟨layers_{input} l l' N = isInput_s (layers_{edges} l l' N)⟩
definition ⟨layers_{output} l l' N = isOutput_s (layers_{edges} l l' N)⟩
definition ⟨layers_{dense} l l' N = isDense_s (layers_{edges} l l' N)⟩
definition ⟨layers_{activation} l l' N = isActivation_s (layers_{edges} 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)
 ⟨proof⟩

lemma *uid_is_singleton*: ⟨x ∈ NN_Digraph.uid ' (neurons N)
 ⇒ is_singleton {n ∈ neurons N. NN_Digraph.uid n = x}⟩
 ⟨proof⟩

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)⟩
 ⟨proof⟩

lemma *output_layer_ids_subset_neuron_ids*: ⟨output_layer_ids N ⊆ uid ' (neurons N)⟩
 ⟨proof⟩

end

Activation layer proofs **lemma** *distinct_activation_edges*: ⟨distinct (activation N φ' α' β')⟩
 ⟨proof⟩

lemma *output_activation_layer_length_equal*:
 assumes *notEmptyNeurons*: ⟨neurons N ≠ {}⟩
 and *notEmptyActivationLayer*: ⟨length(activation N φ' α' β') ≠ 0⟩
 shows ⟨card(output_layer_ids N) = length(activation N φ' α' β')⟩
 ⟨proof⟩

lemma *new_ids_activation_layer_length_equal*:
 assumes *notEmptyNeurons*: ⟨neurons N ≠ {}⟩
 and *notEmptyActivationLayer*: ⟨length(activation N φ' α' β') ≠ 0⟩
 and *notEmptyNewIds*: ⟨length(mk_new_ids N) ≠ 0⟩

shows $\langle \text{length}(\text{mk_new_ids } N) = \text{length}(\text{activation } N \varphi' \alpha' \beta') \rangle$
 $\langle \text{proof} \rangle$

lemma *map_neuron_hd_id*:
 $\langle (\text{map } (\lambda x. \text{Neuron } (\varphi = \varphi', \alpha = \alpha', \beta = \beta', \text{uid} = f x))) X =$
 $(\text{map } (\lambda x. \text{Neuron } (\varphi = \varphi', \alpha = \alpha', \beta = \beta', \text{uid} = x))) (\text{map } f X) \rangle$
 $\langle \text{proof} \rangle$

lemma *map_neuron_tl_id*:
 $\langle (\text{map } (\lambda x. \text{the_elem } \{n \in \text{neurons } N. \text{NN_Digraph.uid } n = f x\}) X =$
 $(\text{map } (\lambda x. \text{the_elem } \{n \in \text{neurons } N. \text{NN_Digraph.uid } n = x\})) (\text{map } f X) \rangle$
 $\langle \text{proof} \rangle$

context *nn_pregraph begin*

lemma *distinct_head_activation*: $\langle \text{distinct}(\text{map } \text{hd } (\text{activation } N \varphi' \alpha' \beta')) \rangle$
 $\langle \text{proof} \rangle$

end

context *neural_network_digraph begin*

lemma *distinct_tail_activation*: $\langle \text{distinct}(\text{map } \text{tl } (\text{activation } N \varphi' \alpha' \beta')) \rangle$
 $\langle \text{proof} \rangle$

lemma *activation_is_activation*: $\langle \text{isActivation}_l(\text{activation } N \varphi' \alpha' \beta') \rangle$
 $\langle \text{proof} \rangle$

end

Output layer proofs **lemma** *output_output_layer_length_equal*:

assumes *notEmptyNeurons*: $\langle \text{neurons } N \neq \{\} \rangle$
and *notEmptyOutputLayer*: $\langle \text{length}(\text{out } N) \neq 0 \rangle$
shows $\langle \text{card}(\text{output_layer_ids } N) = \text{length}(\text{out } N) \rangle$
 $\langle \text{proof} \rangle$

lemma *new_ids_output_layer_length_equal*:
assumes *notEmptyNeurons*: $\langle \text{neurons } N \neq \{\} \rangle$
and *notEmptyOutputLayer*: $\langle \text{length}(\text{out } N) \neq 0 \rangle$
shows $\langle \text{length}(\text{mk_new_ids } N) = \text{length}(\text{out } N) \rangle$
 $\langle \text{proof} \rangle$

lemma *distinct_output_edges*: $\langle \text{distinct}(\text{out } N) \rangle$
 $\langle \text{proof} \rangle$

lemma *map_out_neuron_hd_id*: $\langle (\text{map } (\lambda x. \text{Out } (f x))) X = (\text{map } (\lambda x. \text{Out } x) (\text{map } f X)) \rangle$
 $\langle \text{proof} \rangle$

context *nn_pregraph begin*

lemma *distinct_head_output*: $\langle \text{distinct}(\text{map } \text{hd } (\text{out } N)) \rangle$
 $\langle \text{proof} \rangle$

end

lemma *fold_and_map*: $\langle \text{foldr } (\wedge) (\text{map } (\lambda x. \text{True}) X) \text{True} \rangle$
 $\langle \text{proof} \rangle$

lemma *head_output_neurons*: $\langle \text{foldr } (\wedge) (\text{map } (\text{output_neuron} \circ \text{edge.hd}) (\text{out } N)) \text{True} \rangle$
 $\langle \text{proof} \rangle$

context *neural_network_digraph* **begin**

lemma *distinct_tail_output*: $\langle \text{distinct}(\text{map } \text{tl } (\text{out } N)) \rangle$
 $\langle \text{proof} \rangle$

lemma *output_is_output*: $\langle \text{isOutput}_l (\text{out } N) \rangle$
 $\langle \text{proof} \rangle$

Dense layer proofs **lemma** *dense_is_dense*:
assumes *neuronsNotZero*: $\langle n > 0 \rangle$
and *weightEqualNeurons*: $\langle \text{length } \omega' = n \rangle$
shows $\langle \text{isDense}_s(\text{set}(\text{dense } N n \omega' \varphi' \alpha' \beta')) \rangle$
 $\langle \text{proof} \rangle$

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 $\langle \text{valid_activation_tab}_l \text{ tab} = (\forall f \in \text{ran } \text{tab}. \forall xs. \text{length } xs = \text{length } (f \text{ xs})) \rangle$

lemma *valid_activation_preserves_length*:
assumes $\langle \text{valid_activation_tab}_l \text{ t} \rangle$
assumes $\langle \text{t } n = \text{Some } f \rangle$
shows $\langle \text{length } xs = \text{length } (f \text{ xs}) \rangle$
 $\langle \text{proof} \rangle$

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

$\langle \text{layer_consistent}_l \text{ _ } nc (\text{In } l) = (o < \text{units } l \wedge nc = \text{units } l) \rangle$
 $\mid \langle \text{layer_consistent}_l \text{ _ } nc (\text{Out } l) = (o < \text{units } l \wedge nc = \text{units } l) \rangle$
 $\mid \langle \text{layer_consistent}_l \text{ } N \text{ } nc (\text{Activation } l) = ((o < \text{units } l \wedge nc = \text{units } l) \wedge ((\text{activation_tab } N) (\varphi \text{ } l) \neq \text{None})) \rangle$
 $\mid \langle \text{layer_consistent}_l \text{ } N \text{ } nc (\text{Dense } l) = (o < \text{units } l \wedge o < nc \wedge \text{length } (\beta \text{ } l) = \text{units } l \wedge \text{length } (\omega \text{ } l) = \text{units } l \wedge (\forall r \in \text{set } (\omega \text{ } l). \text{length } r = nc) \rangle$

$\wedge (((\text{activation_tab } N) (\varphi l)) \neq \text{None})) \rangle$

fun *layers_consistent_l* **where**

$\langle \text{layers_consistent}_l N _ [] = \text{True} \rangle$

$\langle \text{layers_consistent}_l N w (l \# ls) = ((\text{layer_consistent}_l N w l) \wedge (\text{layers_consistent}_l N (\text{out_deg_layer } l) ls)) \rangle$

lemma *layer_consistent_l_in_deg_layer*:

assumes *layer_consistent_l N nc l*

shows *in_deg_layer l = nc*

$\langle \text{proof} \rangle$

lemma *layers_consistent_l_in_deg*:

assumes (*layers_consistent_l N nc (l # ls')*)

shows *in_deg_layer l = nc*

$\langle \text{proof} \rangle$

lemma *layer_consistent_l_activation_tab_const*:

$\langle \text{layer_consistent}_l N nc l = \text{layer_consistent}_l (\!| \text{layers} = ls, \text{activation_tab} = \text{activation_tab } N \!|) nc \! \rangle$

$\langle \text{proof} \rangle$

lemma *layers_consistent_l_activation_tab_const*:

$\langle \text{layers_consistent}_l N nc ls = \text{layers_consistent}_l (\!| \text{layers} = ls', \text{activation_tab} = \text{activation_tab } N \!|) nc \! \rangle$

$\langle \text{proof} \rangle$

lemma *layers_consistent_l_layersN_const*:

$\langle \text{layers_consistent}_l N = \text{layers_consistent}_l (\!| \text{layers} = ls', \text{activation_tab} = \text{activation_tab } N \!|) \rangle$

$\langle \text{proof} \rangle$

lemma *layers_consistent_lAll*:

assumes $\langle \text{layers_consistent}_l N \text{ inputs } (\text{layers } N) \rangle$

shows $\langle \forall l \in \text{set } (\text{layers } N). \exists n . \text{layer_consistent}_l N n \! \rangle$

$\langle \text{proof} \rangle$

lemma *layers_consistent_lAll'*:

assumes $\langle \text{layers_consistent}_l N (\text{in_deg_NN } N) (\text{layers } N) \rangle$

shows $\langle \forall l \in \text{set } (\text{layers } N). \exists n . \text{layer_consistent}_l N n \! \rangle$

$\langle \text{proof} \rangle$

lemma *layers_consistent_l_layer_consistent_l_Dense*:

assumes $\langle \text{layers_consistent}_l N (\text{in_deg_NN } N) (\text{layers } N) \rangle$

and $\langle \text{Dense } x3 \in \text{set } (\text{layers } N) \rangle$

shows $\langle \text{layer_consistent}_l N (\text{length } (\omega x3 \! \circ)) (\text{Dense } x3) \rangle$

$\langle \text{proof} \rangle$

lemma *layers_consistent_l_layer_consistent_l_Activation*:

assumes $\langle \text{layers_consistent}_l N (\text{in_deg_NN } N) (\text{layers } N) \rangle$

and $\langle \text{Activation } x3 \in \text{set } (\text{layers } N) \rangle$

shows $\langle \text{layer_consistent}_l N (\text{units } x3) (\text{Activation } x3) \rangle$

$\langle \text{proof} \rangle$

```

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 ≠ []⟩
  ⟨proof⟩

lemma min_length_layers_two: ⟨1 < length (layers N)⟩
  ⟨proof⟩

lemma layers_structure: ⟨∃ il ol ls. layers N = (In il)#ls@[Out ol]⟩
  ⟨proof⟩

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 are core building blocks. For our representation to be a well-formed sequential model, we require that the first layer is an input layer and the last layer is an output layer

```

fun predictlayer_l ::⟨('a::{monoid_add,times} list, 'b, 'a list list) neural_network_seq_layers ⇒ ('a list) option ⇒ ('a list, 'b, 'a list list) layer ⇒ ('a list) 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 N None = None⟩
lemma length_out: ⟨predictlayer_l N' vs (Out l) = Some res ⇒ length(res) = (units l)⟩
  ⟨proof⟩

```

```

fun
  predictlayer_l_impl ::⟨('a::{monoid_add,times} list, 'b, 'a list list) neural_network_seq_layers ⇒ 'a list ⇒ ('a list, 'b, 'a list list) layer ⇒ 'a 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 ⟨predict_{seq_layer_l} N inputs = foldl (predict_{layer_l} N) (Some inputs) (layers N)⟩

definition ⟨predict_{seq_layer_l}_impl N inputs = foldl (predict_{layer_l}_impl N) inputs (layers N)⟩

lemma predict_layer_Some:

```
assumes ⟨layer_consistentl N (length xs) l⟩
shows ⟨predictlayer_l N (Some xs) l ≠ None⟩
⟨proof⟩
```

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⟩
⟨proof⟩

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**

```
⟨predictlayer_l N x xs = predictlayer_l N' x xs⟩
⟨proof⟩
```

lemma predict_{layer_l}_activation_tab_const: ⟨predict_{layer_l} N = predict_{layer_l} (|layers = l, activation_tab = activation_tab N)|⟩

⟨proof⟩

lemma input_layer:

```
assumes ⟨y = length i⟩ and ⟨o < y⟩
shows ⟨predictlayer_l N (Some i) (In (|name = x, units = y)) = (Some i)⟩
⟨proof⟩
```

lemma *output_layer*:

assumes $\langle y = \text{length } i \rangle$ and $\langle o < y \rangle$

shows $\langle \text{predict}_{\text{layer}_l} N (\text{Some } i) (\text{Out } (\text{name} = x, \text{units} = y)) = (\text{Some } i) \rangle$

$\langle \text{proof} \rangle$

lemma *dense_layer*:

shows $\langle \text{predict}_{\text{layer}_l} N (\text{Some } i) (\text{Dense } (\text{name} = x, \text{units} = y, \text{ActivationRecord}.\varphi = p, \text{LayerRecord}.\beta = b, \omega = w)) \rangle$

=

$(\text{if } \text{layer_consistent}_l N (\text{length } i) (\text{Dense } (\text{name} = x, \text{units} = y, \text{ActivationRecord}.\varphi = p, \text{LayerRecord}.\beta = b, \omega = w)) \text{ then}$

$(\text{let } \text{in_w_pairs} = \text{map } (\lambda e. \text{zip } i e) w;$
 $\text{wsums} = \text{map } (\lambda vs'. \sum (x,y) \leftarrow vs'. x*y) \text{ in_w_pairs};$
 $\text{wsum_bias} = \text{map } (\lambda (s,b). s+b) (\text{zip } \text{wsums } b)$

in

$(\text{case } \text{activation_tab } N p \text{ of}$

$\text{None} \Rightarrow \text{None}$

$| \text{Some } f \Rightarrow \text{Some } (f \text{ wsum_bias})))$

$\text{else None}) \rangle$

$\langle \text{proof} \rangle$

lemma *dense_layer'*:

assumes $\langle \text{activation_tab } N p = \text{Some } a \rangle$

shows $\langle \text{predict}_{\text{layer}_l} N (\text{Some } i) (\text{Dense } (\text{name} = x, \text{units} = y, \text{ActivationRecord}.\varphi = p, \text{LayerRecord}.\beta = b, \omega = w)) \rangle$

=

$(\text{if } \text{layer_consistent}_l N (\text{length } i) (\text{Dense } (\text{name} = x, \text{units} = y, \text{ActivationRecord}.\varphi = p, \text{LayerRecord}.\beta = b, \omega = w)) \text{ then}$

$(\text{let } \text{in_w_pairs} = \text{map } (\lambda e. \text{zip } i e) w;$
 $\text{wsums} = \text{map } (\lambda vs'. \sum (x,y) \leftarrow vs'. x*y) \text{ in_w_pairs};$
 $\text{wsum_bias} = \text{map } (\lambda (s,b). s+b) (\text{zip } \text{wsums } b)$

$\text{in } \text{Some } (a \text{ wsum_bias}))$

$\text{else None}) \rangle$

$\langle \text{proof} \rangle$

lemma *activation_layer*:

assumes $\langle y = \text{length } i \rangle$

shows $\langle \text{predict}_{\text{layer}_l} N (\text{Some } i) (\text{Activation } (\text{name} = x, \text{units} = y, \text{ActivationRecord}.\varphi = p)) =$

$(\text{if } \text{layer_consistent}_l N (\text{length } i) (\text{Activation } (\text{name} = x, \text{units} = y, \text{ActivationRecord}.\varphi = p)) \text{ then}$

$(\text{case } \text{activation_tab } N p \text{ of } \text{None} \Rightarrow \text{None} | \text{Some } f \Rightarrow \text{Some } (f i))$

$\text{else None}) \rangle$

$\langle \text{proof} \rangle$

lemma *activation_layer'*:

assumes $\langle y = \text{length } i \rangle$

and $\langle \text{activation_tab } N p = \text{Some } a \rangle$

shows $\langle \text{predict}_{\text{layer}_l} N (\text{Some } i) (\text{Activation } (\text{name} = x, \text{units} = y, \text{ActivationRecord}.\varphi = p)) =$

$(\text{if } \text{layer_consistent}_l N (\text{length } i) (\text{Activation } (\text{name} = x, \text{units} = y, \text{ActivationRecord}.\varphi = p)) \text{ then } \text{Some } (a i) \text{ else } \text{None}) \rangle$

$\langle \text{proof} \rangle$

lemma *predict_layer_l_impl_activation_tab_const*: $\langle \text{predict}_{\text{layer}_l} \text{impl } N = \text{predict}_{\text{layer}_l} \text{impl } (\text{layers} = l, \text{activation_tab} = \text{activation_tab } N) \rangle$

$\langle \text{proof} \rangle$

context *neural_network_sequential_layers* **begin**

lemma *img_None_1*: **assumes** $\langle (\text{predict}_{\text{seq_layer_l}} N \text{ xs}) \neq \text{None} \rangle$ **shows** $\langle (\text{length xs} = (\text{in_deg_NN } N)) \rangle$
<proof>

lemma *img_None_2'*:
assumes *ao*: $\langle \text{layers}' \neq [] \rangle$
and *a4*: $\langle \text{valid_activation_tab}_l \text{ activation_tab}' \rangle$
and *a1*: $\langle \text{layers_consistent}_l (\text{layers} = [], \text{activation_tab} = \text{activation_tab}') \rangle (\text{length xs}) \text{ layers}'$
shows $\langle \text{foldl} (\text{predict}_{\text{layer_l}} (\text{layers} = [], \text{activation_tab} = \text{activation_tab}')) (\text{Some xs}) \text{ layers}' \neq \text{None} \rangle$
<proof>

lemma *img_None_2*:
assumes $\langle \text{length xs} = \text{in_deg_NN } N \rangle$
shows $\langle (\text{predict}_{\text{seq_layer_l}} N \text{ xs}) \neq \text{None} \rangle$
<proof>

lemma *img_None*: $\langle (\text{predict}_{\text{seq_layer_l}} N \text{ xs}) \neq \text{None} \rangle = (\text{length xs} = \text{in_deg_NN } N)$
<proof>

lemma *img_Some*: $\langle (\exists y. (\text{predict}_{\text{seq_layer_l}} N \text{ xs}) = \text{Some } y) \rangle = (\text{length xs} = \text{in_deg_NN } N)$
<proof>

lemma *img_length*: $\langle (\exists y. ((\text{predict}_{\text{seq_layer_l}} N \text{ xs}) = \text{Some } y) \longrightarrow (\text{length } y = \text{out_deg_NN } N)) \rangle$
<proof>

lemma *predict_layer_l_impl_eq*:
assumes $\langle \text{layer_consistent}_l N (\text{length inputs}) l \rangle$
shows $\langle \text{predict}_{\text{layer_l}} N (\text{Some inputs}) l = \text{Some} (\text{predict}_{\text{layer_l_impl}} N \text{ inputs } l) \rangle$
<proof>

lemma *aux_length*: \langle
 $o < \text{units } x3 \implies \text{valid_activation_tab}_l \text{ atab} \implies$
 $\text{inputs} \neq [] \implies$
 $\text{length } (\beta x3) = \text{units } x3 \implies$
 $\text{length } (\omega x3) = \text{units } x3 \implies$
 $\forall r \in \text{set } (\omega x3). \text{length } r = \text{length inputs} \implies$
 $\text{atab } (\varphi x3) = \text{Some } y \implies$
 $(\text{length } (y (\text{map2 } (+) (\text{map } ((\lambda vs'. \sum (x, y) \leftarrow vs'. x * y) \circ \text{zip inputs}) (\omega x3)) (\beta x3)))) = \text{units } x3$
 \rangle
<proof>

lemma *pred_list_impl_aux'*:
assumes $\langle \text{ls} \neq [] \rangle$
and *layer_valid*: $\langle \text{layers_consistent}_l (\text{layers} = [], \text{activation_tab} = \text{atab}) (\text{length inputs}) \text{ ls} \rangle$
and *activation_tab_valid*: $\langle \text{valid_activation_tab}_l \text{ atab} \rangle$
shows \langle
 $\text{foldl} (\text{predict}_{\text{layer_l}} (\text{layers} = [], \text{activation_tab} = \text{atab})) (\text{Some inputs}) \text{ ls} =$
 $\text{Some} (\text{foldl} (\text{predict}_{\text{layer_l_impl}} (\text{layers} = [], \text{activation_tab} = \text{atab})) \text{ inputs } \text{ls})$
 \rangle
<proof>

```

lemma pred_list_impl_aux:
  assumes layer_valid:  $\langle \text{layers\_consistent}_l \ (\!| \text{layers} = \text{ls}, \text{activation\_tab} = \text{atab} |) \ (\text{length inputs}) \ \text{ls} \rangle$ 
  and activation_tab_valid:  $\langle \text{valid\_activation\_tab}_l \ \text{atab} \rangle$ 
  shows  $\langle$ 
     $\text{foldl} \ (\text{predict}_{l_{\text{layer}}_l} \ (\!| \text{layers} = \text{ls}, \text{activation\_tab} = \text{atab} |)) \ (\text{Some inputs}) \ \text{ls} =$ 
     $\text{Some} \ (\text{foldl} \ (\text{predict}_{l_{\text{layer}}_l\_impl} \ (\!| \text{layers} = \text{ls}, \text{activation\_tab} = \text{atab} |)) \ \text{inputs} \ \text{ls})$ 
   $\rangle$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma predict_seq_layer_l_code [code]:
  assumes  $\langle \text{in\_deg\_NN} \ N = \text{length inputs} \rangle$ 
  shows  $\langle \text{predict}_{\text{seq\_layer}_l} \ N \ \text{inputs} = \text{Some} \ (\text{predict}_{\text{seq\_layer}_l\_impl} \ N \ \text{inputs}) \rangle$ 
   $\langle \text{proof} \rangle$ 

```

```

lemma predict_seq_layer_l_code' [code]:
  assumes  $\langle \text{in\_deg\_NN} \ N = \text{length inputs} \rangle$ 
  shows  $\langle \text{the} \ (\text{predict}_{\text{seq\_layer}_l} \ N \ \text{inputs}) = \text{predict}_{\text{seq\_layer}_l\_impl} \ N \ \text{inputs} \rangle$ 
   $\langle \text{proof} \rangle$ 

```

end

$\langle ML \rangle$

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 $\langle \text{valid_activation_tab}_m \ t \rangle$

assumes $\langle t \ n = \text{Some } f \rangle$

shows $\langle \text{dim_vec } \text{xs} = \text{dim_vec} \ (f \ \text{xs}) \rangle$

$\langle \text{proof} \rangle$

fun *layer_consistent_m* :: $('a \ \text{vec}, 'b, 'c \ \text{mat}) \ \text{neural_network_seq_layers} \Rightarrow \text{nat} \Rightarrow ('a \ \text{vec}, 'b, 'c \ \text{mat}) \ \text{layer} \Rightarrow \text{bool}$

where

$\langle \text{layer_consistent}_m \ _ \ nc \ (\text{In } l) = (o < \text{units } l \wedge nc = \text{units } l) \rangle$

$\mid \langle \text{layer_consistent}_m \ _ \ nc \ (\text{Out } l) = (o < \text{units } l \wedge nc = \text{units } l) \rangle$

$\mid \langle \text{layer_consistent}_m \ N \ nc \ (\text{Activation } l) = ((o < \text{units } l \wedge nc = \text{units } l)$

$\wedge ((\text{activation_tab } N) \ (\varphi \ l) \neq \text{None})) \rangle$

```

| <layer_consistentm N nc (Dense l) = (o < units l ∧ o < nc
  ∧ dim_vec (β l) = units l
  ∧ dim_col (ω l) = units l
  ∧ dim_row (ω l) = nc
  ∧ ((activation_tab N) (φ l)) ≠ None )>

```

```

fun layers_consistentm where
  <layers_consistentm N [] = True>
| <layers_consistentm N w (l#ls) = ((layer_consistentm N w l) ∧ (layers_consistentm N (out_deg_layer l) ls))>

```

```

lemma layer_consistentm_activation_tab_const:
  <layer_consistentm N nc l = layer_consistentm (layers = ls, activation_tab = activation_tab N) nc l>
  <proof>

```

```

lemma layers_consistentm_activation_tab_const:
  <layers_consistentm N nc ls = layers_consistentm (layers = ls', activation_tab = activation_tab N) nc ls>
  <proof>

```

```

lemma layers_consistentm_All:
  assumes <layers_consistentm N inputs (layers N)>
  shows <∀ l ∈ set (layers N). ∃ n . layer_consistentm N n l>
  <proof>

```

```

lemma layers_consistentm_All':
  assumes <layers_consistentm N (in_deg_NN N) (layers N)>
  shows <∀ l ∈ set (layers N). ∃ n . layer_consistentm N n l>
  <proof>

```

```

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 ≠ []>
  <proof>

```

```

lemma min_length_layers_two: <1 < length (layers N)>
  <proof>

```

```

lemma layers_structure: <∃ il ol ls. layers N = (In il)#ls@[Out ol]>
  <proof>

```

end

```

fun predictlayerm::('a::comm_ring Matrix.vec, 'b, 'a Matrix.mat) neural_network_seq_layers ⇒ ('a Matrix.vec) option
⇒ ('a Matrix.vec, 'b, 'a Matrix.mat) layer ⇒ ('a Matrix.vec) option where
  <predictlayerm N (Some vs) (In l) = (if layer_consistentm N (dim_vec vs) (In l) then Some vs else None) >
| <predictlayerm N (Some vs) (Out l) = (if layer_consistentm N (dim_vec vs) (Out l) then Some vs else None) >
| <predictlayerm N (Some vs) (Dense pl) = (if layer_consistentm N (dim_vec vs) (Dense pl) then

```

```

      (case activation_tab N ( $\varphi$  pl) of
        None  $\Rightarrow$  None
      | Some f  $\Rightarrow$  Some (f ((vs v*  $\omega$  pl) +  $\beta$  pl))
      ) else None ) $\rangle$ 
|  $\langle$ predictlayer_m N (Some vs) (Activation pl) = (if layer_consistentm N (dim_vec vs) (Activation pl) then
      (case activation_tab N ( $\varphi$  pl) of
        None  $\Rightarrow$  None
      | Some f  $\Rightarrow$  Some (f vs)
      ) else None ) $\rangle$ 
|  $\langle$ predictlayer_m None = None  $\rangle$ 

```

fun

```

predictlayer_m_impl ::  $\langle$ ('a::comm_ring} Matrix.vec, 'b, 'a Matrix.mat) neural_network_seq_layers  $\Rightarrow$  'a Matrix.vec
 $\Rightarrow$  ('a Matrix.vec, 'b, 'a Matrix.mat) layer  $\Rightarrow$  'a Matrix.vec

```

where

```

 $\langle$ predictlayer_m_impl N vs (In l) = vs $\rangle$ 
|  $\langle$ predictlayer_m_impl N vs (Out l) = vs $\rangle$ 
|  $\langle$ predictlayer_m_impl N vs (Dense pl) = ((the (activation_tab N ( $\varphi$  pl))) ((vs v*  $\omega$  pl) +  $\beta$  pl)) $\rangle$ 
|  $\langle$ predictlayer_m_impl N vs (Activation pl) = (the (activation_tab N ( $\varphi$  pl)) vs) $\rangle$ 

```

lemma predict_layer_Some:

```

assumes  $\langle$ (layer_consistentm N (dim_vec xs) l) $\rangle$ 
shows  $\langle$ (predictlayer_m N (Some xs) l  $\neq$  None)  $\rangle$ 
 $\langle$ proof $\rangle$ 

```

definition \langle predict_{seq_layer_m} N inputs = foldl (predict_{layer_m} N) (Some inputs) (layers N) \rangle

definition \langle predict_{seq_layer_m}_impl N inputs = foldl (predict_{layer_m}_impl N) inputs (layers N) \rangle

definition \langle predict_{seq_layer_m}' N inputs = map_option list_of_vec (predict_{seq_layer_m} N (vec_of_list inputs)) \rangle

lemma predict_{layer_l}_impl_activation_tab_const: \langle predict_{layer_m}_impl N = predict_{layer_m}_impl (layers = l, activation_tab = activation_tab N) \rangle

\langle proof \rangle

lemma layers_consistent_m_layersN_const:

```

 $\langle$ layers_consistentm N = layers_consistentm (layers = ls', activation_tab = activation_tab N) $\rangle$ 
 $\langle$ proof $\rangle$ 

```

lemma predict_{layer_m}_impl_eq:

```

assumes  $\langle$ layer_consistentm N (dim_vec inputs) l $\rangle$ 
shows  $\langle$ predictlayer_m N (Some inputs) l = Some (predictlayer_m_impl N inputs l) $\rangle$ 
 $\langle$ proof $\rangle$ 

```

lemma valid_activation_preserves_length:

```

assumes  $\langle$ valid_activation_tabm t $\rangle$ 
assumes  $\langle$ t n = Some f $\rangle$ 
shows  $\langle$ dim_vec xs = dim_vec (f xs) $\rangle$ 
 $\langle$ proof $\rangle$ 

```

lemma fold_predict_m_strict: \langle (foldl (predict_{layer_m} N) None ls) = None \rangle

$\langle proof \rangle$

lemmas $[nn_layer] = predict_{layer_m} \cdot_simps\ predict_layer_Some\ fold_predict_m_strict$

lemma $predict_{layer_m}_activation_tab$: **assumes** $activation_tab\ N = activation_tab\ N'$ **shows**

$\langle predict_{layer_m}\ N\ x\ xs = predict_{layer_m}\ N'\ x\ xs \rangle$

$\langle proof \rangle$

lemma $predict_{layer_m}_activation_tab_const$: $\langle predict_{layer_m}\ N = predict_{layer_m}\ (\!(layers = l, activation_tab = activation_tab\ N)\!) \rangle$

$\langle proof \rangle$

context $neural_network_sequential_layers_m$ **begin**

lemma img_None_1 : **assumes** $\langle (predict_{seq_layer_m}\ N\ xs) \neq None \rangle$ **shows** $\langle (dim_vec\ xs = (in_deg_NN\ N)) \rangle$

$\langle proof \rangle$

lemma img_None_2' :

assumes ao : $\langle layers' \neq [] \rangle$

and $a4$: $\langle valid_activation_tab_m\ activation_tab' \rangle$

and $a1$: $\langle layers_consistent_m\ (\!(layers = [], activation_tab = activation_tab')\!) (dim_vec\ xs)\ layers' \rangle$

shows $\langle foldl\ (predict_{layer_m}\ (\!(layers = [], activation_tab = activation_tab')\!))\ (Some\ xs)\ layers' \neq None \rangle$

$\langle proof \rangle$

lemma img_None_2 :

assumes $\langle dim_vec\ xs = in_deg_NN\ N \rangle$

shows $\langle (predict_{seq_layer_m}\ N\ xs) \neq None \rangle$

$\langle proof \rangle$

lemma img_None : $\langle (predict_{seq_layer_m}\ N\ xs) \neq None = (dim_vec\ xs = in_deg_NN\ N) \rangle$

$\langle proof \rangle$

lemma img_Some : $\langle (\exists y. (predict_{seq_layer_m}\ N\ xs) = Some\ y) = (dim_vec\ xs = in_deg_NN\ N) \rangle$

$\langle proof \rangle$

lemma img_deg : $\langle (\exists y. ((predict_{seq_layer_m}\ N\ xs) = Some\ y) \longrightarrow (dim_vec\ y = out_deg_NN\ N)) \rangle$

$\langle proof \rangle$

lemma aux_length : $0 < units\ x3 \implies$

$0 < dim_vec\ inputs \implies$

$dim_vec\ (\beta\ x3) = units\ x3 \implies$

$dim_col\ (\omega\ x3) = units\ x3 \implies$

$dim_row\ (\omega\ x3) = dim_vec\ inputs \implies$

$layers_consistent_m\ (\!(layers = [], activation_tab = atab)\!) (units\ x3)\ xs \implies$

$atab\ (\varphi\ x3) = Some\ y \implies valid_activation_tab_m\ atab \implies layers_consistent_m\ (\!(layers = [], activation_tab = atab)\!)$

$(dim_vec\ (y\ (inputs\ v * \omega\ x3 + \beta\ x3)))\ xs$

$\langle proof \rangle$

lemma $pred_mat_impl_aux'$:

assumes $\langle ls \neq [] \rangle$

and $layer_valid$: $\langle layers_consistent_m\ (\!(layers = [], activation_tab = atab)\!) (dim_vec\ inputs)\ ls \rangle$

and $activation_tab_valid$: $\langle valid_activation_tab_m\ atab \rangle$

shows \langle

$foldl\ (predict_{layer_m}\ (\!(layers = [], activation_tab = atab)\!))\ (Some\ inputs)\ ls =$

$Some\ (foldl\ (predict_{layer_m}_impl\ (\!(layers = [], activation_tab = atab)\!))\ inputs\ ls)$

```
>  
<proof>
```

```
lemma pred_mat_impl_aux:  
  assumes layer_valid: <layers_consistent_m (layers = ls, activation_tab = atab) (dim_vec inputs) ls>  
    and activation_tab_valid: <valid_activation_tab_m atab>  
  shows <  
    foldl (predict_layer_m (layers = ls, activation_tab = atab)) (Some inputs) ls =  
    Some (foldl (predict_layer_m_impl (layers = ls, activation_tab = atab)) inputs ls)  
  >  
<proof>
```

```
lemma predict_seq_layer_m_code [code]:  
  assumes <in_deg_NN N = dim_vec inputs>  
  shows <predict_seq_layer_m N inputs = Some (predict_seq_layer_m_impl N inputs)>  
<proof>
```

```
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>  
<proof>
```

end

<ML>

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 l, units l, φ = φ l))>  
  | <layer_list_to_matrix (Dense l) = (let dimc = length (List.hd (ω l)) in  
    (Dense (name l, units l, φ = φ l,
```

$$\beta = \text{vec_of_list } (\beta \ l), \omega = \text{transpose_mat } (\text{mat_of_rows_list } \text{dimc } (\omega \ l)) \ \! \! \! \rangle \rangle$$

fun

`layer_matrix_to_list::('a Matrix.vec, 'b, 'a Matrix.mat) layer \Rightarrow ('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, $\varphi = \varphi \ l$)>`

`| <layer_matrix_to_list (Dense l) = (Dense (|name = name l, units = units l, $\varphi = \varphi \ l$,
 $\beta = \text{list_of_vec } (\beta \ l), \omega = \text{mat_to_list } (\text{transpose_mat } (\omega \ l)) \ \! \! \! \rangle \rangle$)>`

definition `activation_list_to_matrix::('b \Rightarrow (('a list \Rightarrow 'a list) option)) \Rightarrow ('b \Rightarrow (('a Matrix.vec \Rightarrow 'a Matrix.vec) option))`

where

`activation_list_to_matrix a = map_option ($\lambda f . \text{vec_of_list } \circ f \circ \text{list_of_vec}$) \circ a`

definition `activation_matrix_to_list::('b \Rightarrow (('a Matrix.vec \Rightarrow 'a Matrix.vec) option)) \Rightarrow ('b \Rightarrow (('a list \Rightarrow 'a list) option))`

where

`activation_matrix_to_list a = map_option ($\lambda f . \text{list_of_vec } \circ f \circ \text{vec_of_list}$) \circ a`

definition

`nn_list_to_matrix::('a list, 'b, 'a list list) neural_network_seq_layers \Rightarrow ('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 \Rightarrow ('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 \implies layer_matrix_to_list (layer_list_to_matrix l) = l
<proof>`

lemma `layer_list_list_inverse:`

`<layer_consistentm N n l \implies layer_list_to_matrix (layer_matrix_to_list l) = l
<proof>`

lemma `activation_list_inverse:` `<activation_matrix_to_list (activation_list_to_matrix a) x = a x>`

`<proof>`

lemma `activation_list_inverse':` `<activation_matrix_to_list (activation_list_to_matrix a) = a>`

`<proof>`

lemma *activation_matrix_inverse*: $\langle \text{activation_list_to_matrix } (\text{activation_matrix_to_list } a) x = a x \rangle$
<proof>

lemma *activation_matrix_inverse'*: $\langle \text{activation_list_to_matrix } (\text{activation_matrix_to_list } a) = a \rangle$
<proof>

lemma *is_In_seq_L_eq_m*:
assumes $\langle (\text{layers } N) \neq [] \rangle$
shows $\langle \text{isIn } (\text{List.hd } (\text{layers } N)) = \text{isIn } (\text{List.hd } (\text{layers } (\text{nn_list_to_matrix } N))) \rangle$
<proof>

lemma *is_Out_seq_L_eq_m*:
assumes $\langle (\text{layers } N) \neq [] \rangle$
shows $\langle \text{isOut } (\text{last } (\text{layers } N)) = \text{isOut } (\text{last } (\text{layers } (\text{nn_list_to_matrix } N))) \rangle$
<proof>

lemma *is_Internal_seq_L_eq_m*:
assumes $\langle (\text{layers } N) \neq [] \rangle$
shows $\langle \text{list_all isInternal } ((\text{List.tl o butlast}) (\text{layers } N)) = \text{list_all isInternal } ((\text{List.tl o butlast}) (\text{layers } (\text{nn_list_to_matrix } N))) \rangle$
<proof>

lemma *valid_activation_tab_seq_L_imp_m*:
 $\langle \text{valid_activation_tab}_l (\text{activation_tab } N) \implies \text{valid_activation_tab}_m (\text{activation_tab } (\text{nn_list_to_matrix } N)) \rangle$
<proof>

lemma *layers_consistent_seq_L_imp_m*:
assumes $\langle \text{layers_consistent}_l N n (\text{layers } N) \rangle$
shows $\langle \text{layers_consistent}_m (\text{nn_list_to_matrix } N) n (\text{layers } (\text{nn_list_to_matrix } N)) \rangle$
<proof>

lemma *in_deg_seq_L_eq_m*: $\langle \text{in_deg_NN } N = (\text{in_deg_NN } (\text{nn_list_to_matrix } N)) \rangle$
<proof>

lemma *is_In_seq_m_eq_l*:
assumes $\langle (\text{layers } N) \neq [] \rangle$
shows $\langle \text{isIn } (\text{List.hd } (\text{layers } N)) = \text{isIn } (\text{List.hd } (\text{layers } (\text{nn_matrix_to_list } N))) \rangle$
<proof>

lemma *is_Out_seq_m_eq_l*:
assumes $\langle (\text{layers } N) \neq [] \rangle$
shows $\langle \text{isOut } (\text{last } (\text{layers } N)) = \text{isOut } (\text{last } (\text{layers } (\text{nn_matrix_to_list } N))) \rangle$
<proof>

lemma *is_Internal_seq_m_eq_l*:
assumes $\langle (\text{layers } N) \neq [] \rangle$
shows $\langle \text{list_all isInternal } ((\text{List.tl o butlast}) (\text{layers } N)) = \text{list_all isInternal } ((\text{List.tl o butlast}) (\text{layers } (\text{nn_matrix_to_list } N))) \rangle$
<proof>

lemma *valid_activation_tab_seq_m_imp_l*:
 $\langle \text{valid_activation_tab}_m (\text{activation_tab } N) \implies \text{valid_activation_tab}_l (\text{activation_tab } (\text{nn_matrix_to_list } N)) \rangle$
<proof>

lemma *layers_consistent_seq_m_imp_l*:
assumes $\langle \text{layers_consistent}_m N n \text{ (layers } N) \rangle$
shows $\langle \text{layers_consistent}_l (\text{nn_matrix_to_list } N) n \text{ (layers (nn_matrix_to_list } N)) \rangle$
 $\langle \text{proof} \rangle$

lemma *in_deg_seq_m_eq_l*: $\langle \text{in_deg_NN } N = (\text{in_deg_NN } (\text{nn_matrix_to_list } N)) \rangle$
 $\langle \text{proof} \rangle$

theorem *neural_network_sequential_L_m*:
 $\langle \text{neural_network_sequential_layers}_l N \implies \text{neural_network_sequential_layers}_m (\text{nn_list_to_matrix } N) \rangle$
 $\langle \text{proof} \rangle$

theorem *neural_network_sequential_m_l*:
 $\langle \text{neural_network_sequential_layers}_m N \implies \text{neural_network_sequential_layers}_l (\text{nn_matrix_to_list } N) \rangle$
 $\langle \text{proof} \rangle$

lemma *matrix_list_inverse*:
assumes $\langle \text{layers_consistent}_l N n \text{ (layers } N) \rangle$
shows $\langle \text{nn_matrix_to_list } (\text{nn_list_to_matrix } N) = N \rangle$
 $\langle \text{proof} \rangle$

lemma *list_matrix_inverse*:
assumes $\langle \text{layers_consistent}_m N n \text{ (layers } N) \rangle$
shows $\langle \text{nn_list_to_matrix } (\text{nn_matrix_to_list } N) = N \rangle$
 $\langle \text{proof} \rangle$

lemma *square_nth_nth_id*:
 $\forall w \in \text{set } ws. \text{length } w = \text{length } ws \implies$
 $(\text{map } (\lambda i. (\text{map } (\lambda ia. ws ! i ! ia) [0..<\text{length } ws]))) [0..<\text{length } ws] = ws$
 $\langle \text{proof} \rangle$

lemma *nth_map_f*: $\langle \text{map } ((\lambda i. f(xs ! i))) [0..<\text{length } xs] = \text{map } f xs \rangle$
 $\langle \text{proof} \rangle$

lemma *square_nth_nth_id_f*:
 $\forall w \in \text{set } ws. \text{length } w = \text{length } ws \implies$
 $(\text{map } (\lambda i. (\text{map } (\lambda ia. f (ws ! i ! ia)) [0..<\text{length } ws]))) [0..<\text{length } ws] = \text{map } (\text{map } f) ws$
 $\langle \text{proof} \rangle$

lemma *F*: $\langle \text{length } (ws::'a::\{\text{comm_ring}\} \text{ list}) = \text{length } \text{Inputs} \implies \text{map } (\lambda ia. ws ! ia * \text{Inputs} ! ia) [0..<\text{length } \text{Inputs}] = \text{map2 } (*) \text{Inputs } ws \rangle$
 $\langle \text{proof} \rangle$

lemma *list_singleton*: $\langle \text{length } xs = 1 \implies \exists e. xs = [e] \rangle$
 $\langle \text{proof} \rangle$

lemma *activation_list_to_matrix_eq*:
 $\langle \text{predict}_{\text{layer}_l} N (\text{Some } (vs::'a::\text{comm_ring} \text{ list})) (\text{Activation } pl) =$
 $\text{map_option } \text{list_of_vec } (\text{predict}_{\text{layer}_m} (\text{nn_list_to_matrix } N) (\text{Some } (\text{vec_of_list } vs))) ((\text{layer_list_to_matrix$

(Activation pl)))) ›
⟨proof⟩

lemma layers_matrix_to_list:
⟨layers (nn_matrix_to_list N) = map layer_matrix_to_list (layers N)›
⟨proof⟩

lemma layers_list_to_matrix:
⟨layers (nn_list_to_matrix N) = map layer_list_to_matrix (layers N)›
⟨proof⟩

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)›
⟨proof⟩

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))›
⟨proof⟩

lemma layers_list_to_matrix_none:
⟨activation_tab N p = None ⇒ (activation_tab (nn_list_to_matrix N)) p = None›
⟨proof⟩

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)))›
›
⟨proof⟩

lemma activation_list_to_matrix:
⟨(activation_tab (nn_list_to_matrix N)) = (activation_list_to_matrix (activation_tab N))›
⟨proof⟩

lemma vec_add_list:
assumes ⟨dim_vec M = length bs⟩
shows ⟨M + vec_of_list bs = vec_of_list (map2 (+) (list_of_vec M) bs)›
⟨proof⟩

lemma vec_add_list':
assumes ⟨dim_vec M = dim_vec bs⟩
shows ⟨M + bs = vec_of_list (map2 (+) (list_of_vec M) (list_of_vec bs))›
⟨proof⟩

lemma list_of_vec_map':
⟨v = vec_of_list (map ((vec_index) v) [0..<dim_vec v])›
⟨proof⟩

lemma mat_list_transpose:
assumes ⟨0 < dim_row M⟩ and ⟨0 < dim_col M⟩
shows ⟨(mat_to_list M^T) = List.transpose (mat_to_list M)›
⟨proof⟩

lemma dim_row_mat_not_zero:

assumes $\langle \text{dim_row } M \neq 0 \rangle$
shows $\langle \text{mat_to_list } M \neq [] \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{map2_to_map_idx_eq}$: $\langle \text{length } xs = \text{length } ys \implies (\text{map2 } (*) \text{ } xs \text{ } (ys)) = \text{map } (\lambda i. xs!i * ys!i) [0..< \text{length } xs] \rangle$
 $\langle \text{proof} \rangle$

lemma map2_to_map_idx : $\langle (\text{map2 } (*) \text{ } xs \text{ } (ys)) = \text{map } (\lambda i. xs!i * ys!i) [0..< \min (\text{length } xs) (\text{length } ys)] \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{length_list_transpose_mat}$: $\langle 0 < \text{dim_row } M \implies 0 < \text{dim_col } M \implies \text{length } (\text{List.transpose } (\text{mat_to_list } M)) = \text{dim_col } M \rangle$
 $\langle \text{proof} \rangle$

lemma map_sum_list_idx : \langle
 $\text{map } (\lambda m. \text{sum_list } (\text{map2 } (*) \text{ } m \text{ } (\text{list_of_vec } v))) (\text{List.transpose } (\text{mat_to_list } M))$
 $= \text{map } (\lambda i. \text{sum_list } (\text{map2 } (*) \text{ } ((\text{List.transpose } (\text{mat_to_list } M))!i) \text{ } (\text{list_of_vec } v))) [0..< \text{length } (\text{List.transpose } (\text{mat_to_list } M))]$
 $\langle \text{proof} \rangle$

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 $\langle (v :: 'a :: \text{comm_ring } \text{vec}) \cdot v * M = \text{vec_of_list } (\text{map } (\lambda m. \text{sum_list } (\text{map2 } (*) \text{ } m \text{ } (\text{list_of_vec } v))) (\text{mat_to_list } M^T)) \rangle$
 $\langle \text{proof} \rangle$

lemma hd_length_inputs : $\langle 0 < \text{units } x3 \implies$
 $\text{length } (\beta \text{ } x3) = \text{units } x3 \implies \text{length } (\omega \text{ } x3) = \text{units } x3 \implies \forall r \in \text{set } (\omega \text{ } x3). \text{length } r = \text{length } \text{Inputs} \implies \text{length } \text{Inputs}$
 $= \text{length } (\text{List.hd } (\omega \text{ } x3)) \rangle$
 $\langle \text{proof} \rangle$

5.3.3 Semantic Equivalence of List-based and Matrix-based Models

lemma In_l_to_m_eq :
 $\langle \text{predict}_{\text{layer_l}} N (\text{Some } vs) (\text{In } l) = \text{map_option } \text{list_of_vec } (\text{predict}_{\text{layer_m}} (\text{nn_list_to_matrix } N) (\text{Some } (\text{vec_of_list } vs))) (\text{layer_list_to_matrix } (\text{In } l)) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{In_l_to_m_eq}'$:
 $\langle \text{predict}_{\text{layer_m}} (\text{nn_list_to_matrix } N) (\text{Some } (\text{vec_of_list } vs)) (\text{layer_list_to_matrix } (\text{In } l)) = \text{map_option } \text{vec_of_list } (\text{predict}_{\text{layer_l}} N (\text{Some } vs) (\text{In } l)) \rangle$
 $\langle \text{proof} \rangle$

lemma Out_l_to_m_eq :
 $\langle \text{predict}_{\text{layer_l}} N (\text{Some } vs) (\text{Out } l) = \text{map_option } \text{list_of_vec } (\text{predict}_{\text{layer_m}} (\text{nn_list_to_matrix } N) (\text{Some } (\text{vec_of_list } vs))) (\text{layer_list_to_matrix } (\text{Out } l)) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{Out_l_to_m_eq}'$:
 $\langle \text{predict}_{\text{layer_m}} (\text{nn_list_to_matrix } N) (\text{Some } (\text{vec_of_list } vs)) (\text{layer_list_to_matrix } (\text{Out } l)) = \text{map_option } \text{vec_of_list } (\text{predict}_{\text{layer_l}} N (\text{Some } vs) (\text{Out } l)) \rangle$

<proof>

lemma *Dense_l_to_m_eq*:

assumes $\langle \text{layer_consistent}_l N (\text{length } vs) (\text{Dense } l) \rangle$

shows $\langle \text{predict}_{\text{layer}_l} N (\text{Some } (vs::'a::\text{comm_ring_list})) (\text{Dense } l) \rangle$

$= \text{map_option list_of_vec } (\text{predict}_{\text{layer}_m} (\text{nn_list_to_matrix } N) (\text{Some } (\text{vec_of_list } vs))) (\text{layer_list_to_matrix } (\text{Dense } l)) \rangle$

<proof>

lemma *Dense_l_to_m_eq'*:

assumes $\langle \text{layer_consistent}_l N (\text{length } vs) (\text{Dense } l) \rangle$

shows $\langle \text{predict}_{\text{layer}_m} (\text{nn_list_to_matrix } N) (\text{Some } (\text{vec_of_list } vs)) (\text{layer_list_to_matrix } (\text{Dense } l)) \rangle$

$= \text{map_option vec_of_list } (\text{predict}_{\text{layer}_l} N (\text{Some } (vs::'a::\text{comm_ring_list})) (\text{Dense } l)) \rangle$

<proof>

lemma *Activation_l_to_m_eq*:

$\langle \text{predict}_{\text{layer}_l} N (\text{Some } vs) (\text{Activation } l) \rangle$

$= \text{map_option list_of_vec } (\text{predict}_{\text{layer}_m} (\text{nn_list_to_matrix } N) (\text{Some } (\text{vec_of_list } vs))) (\text{layer_list_to_matrix } (\text{Activation } l)) \rangle$

<proof>

lemma *Activation_l_to_m_eq'*:

$\langle \text{predict}_{\text{layer}_m} (\text{nn_list_to_matrix } N) (\text{Some } (\text{vec_of_list } vs)) (\text{layer_list_to_matrix } (\text{Activation } l)) \rangle$

$= \text{map_option vec_of_list } (\text{predict}_{\text{layer}_l} N (\text{Some } vs) (\text{Activation } l)) \rangle$

<proof>

lemma *aux1*: \langle

$\bigwedge y. l = \text{Dense } x3 \implies$

$(\bigwedge \text{Inputs.}$

$\text{layer_consistent}_l (\text{layers} = l0, \text{activation_tab} = \text{activation_tab}') (\text{length } \text{Inputs}) \text{layers}' \implies$

$\text{foldl } (\text{predict}_{\text{layer}_l} (\text{layers} = l1, \text{activation_tab} = \text{activation_tab}')) (\text{Some } \text{Inputs}) \text{layers}' =$

$\text{map_option list_of_vec } (\text{foldl } (\text{predict}_{\text{layer}_m} (\text{nn_list_to_matrix } (\text{layers} = l2, \text{activation_tab} = \text{activation_tab}'))))$

$(\text{Some } (\text{vec_of_list } \text{Inputs})) (\text{layers } (\text{nn_list_to_matrix } (\text{layers} = \text{layers}', \text{activation_tab} = a2)))) \implies$

$\text{valid_activation_tab}_l \text{activation_tab}' \implies$

$0 < \text{units } x3 \implies$

$\text{Inputs} \neq [] \implies$

$\text{length } (\text{LayerRecord}.\beta x3) = \text{units } x3 \implies$

$\text{length } (\text{LayerRecord}.\omega x3) = \text{units } x3 \implies$

$\forall r \in \text{set } (\text{LayerRecord}.\omega x3). \text{length } r = \text{length } \text{Inputs} \implies$

$\text{layer_consistent}_l (\text{layers} = l0, \text{activation_tab} = \text{activation_tab}') (\text{units } x3) \text{layers}' \implies$

$\text{activation_tab}' (\text{ActivationRecord}.\varphi x3) = \text{Some } y \implies$

$\text{foldl } (\text{predict}_{\text{layer}_l} (\text{layers} = l1, \text{activation_tab} = \text{activation_tab}')) (\text{Some } (y (\text{map2 } (+) (\text{map } ((\lambda vs'. \sum (x, y) \leftarrow vs'. x * y) \circ \text{zip } \text{Inputs}) (\text{LayerRecord}.\omega x3)) (\text{LayerRecord}.\beta x3)))) \text{layers}' =$

$\text{map_option list_of_vec}$

$(\text{foldl } (\text{predict}_{\text{layer}_m} (\text{nn_list_to_matrix } (\text{layers} = l2, \text{activation_tab} = \text{activation_tab}')))) (\text{Some } (\text{vec_of_list } (y (\text{map2 } (+) (\text{map } ((\lambda vs'. \sum (x, y) \leftarrow vs'. x * y) \circ \text{zip } \text{Inputs}) (\text{LayerRecord}.\omega x3)) (\text{LayerRecord}.\beta x3))))))$

$(\text{map } \text{layer_list_to_matrix } \text{layers}') \rangle$

<proof>

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 $\langle \text{foldl} \ (\text{predict}_{\text{layer}_l} \ (\!| \text{layers} = l_1, \text{activation_tab} = \text{activation_tab}' \!)) \ (\text{Some} \ (\text{Inputs})) \ (\text{layers} \ (\!| \text{layers} = \text{layers}', \text{activation_tab} = a_1 \!)) =$
 $\text{map_option list_of_vec}$
 $\ (\text{foldl} \ (\text{predict}_{\text{layer}_m} \ (\text{nn_list_to_matrix} \ (\!| \text{layers} = l_2, \text{activation_tab} = \text{activation_tab}' \!))) \ (\text{Some} \ (\text{vec_of_list} \ \text{Inputs}))$
 $\ (\text{layers} \ (\text{nn_list_to_matrix} \ (\!| \text{layers} = \text{layers}', \text{activation_tab} = a_2 \!))) \rangle$
 $\langle \text{proof} \rangle$

theorem $\text{predict_seq}_l \ \text{eq}_m$:

assumes $\langle \text{layers_consistent}_l \ N \ (\text{length} \ \text{Inputs}) \ (\text{layers} \ N) \rangle$
and $\langle \text{valid_activation_tab}_l \ (\text{activation_tab} \ N) \rangle$
shows $\langle \text{predict}_{\text{seq_layer}_l} \ N \ (\text{Inputs}::'a::\text{comm_ring list}) = \text{predict}_{\text{seq_layer}_m} \ (\text{nn_list_to_matrix} \ N) \ \text{Inputs} \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{In}_m \ \text{to}_l \ \text{eq}$:

$\langle \text{predict}_{\text{layer}_m} \ N \ (\text{Some} \ \text{vs}) \ (\text{In} \ l) = \text{map_option} \ \text{vec_of_list} \ (\text{predict}_{\text{layer}_l} \ (\text{nn_matrix_to_list} \ N) \ (\text{Some} \ (\text{list_of_vec} \ \text{vs}))) \ (\text{layer_matrix_to_list} \ (\text{In} \ l)) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{In}_m \ \text{to}_l \ \text{eq}'$:

$\langle \text{predict}_{\text{layer}_l} \ (\text{nn_matrix_to_list} \ N) \ (\text{Some} \ (\text{list_of_vec} \ \text{vs})) \ (\text{layer_matrix_to_list} \ (\text{In} \ l)) = \text{map_option} \ \text{list_of_vec}$
 $\ (\text{predict}_{\text{layer}_m} \ N \ (\text{Some} \ \text{vs}) \ (\text{In} \ l)) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{Out}_m \ \text{to}_l \ \text{eq}$:

$\langle \text{predict}_{\text{layer}_m} \ N \ (\text{Some} \ \text{vs}) \ (\text{Out} \ l) = \text{map_option} \ \text{vec_of_list} \ (\text{predict}_{\text{layer}_l} \ (\text{nn_matrix_to_list} \ N) \ (\text{Some} \ (\text{list_of_vec} \ \text{vs}))) \ (\text{layer_matrix_to_list} \ (\text{Out} \ l)) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{Out}_m \ \text{to}_l \ \text{eq}'$:

$\langle \text{predict}_{\text{layer}_l} \ (\text{nn_matrix_to_list} \ N) \ (\text{Some} \ (\text{list_of_vec} \ \text{vs})) \ (\text{layer_matrix_to_list} \ (\text{In} \ l)) = \text{map_option} \ \text{list_of_vec}$
 $\ (\text{predict}_{\text{layer}_m} \ N \ (\text{Some} \ \text{vs}) \ (\text{Out} \ l)) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{Dense}_m \ \text{to}_l \ \text{eq}$:

assumes $\langle \text{layer_consistent}_m \ N \ (\text{dim_vec} \ \text{vs}) \ (\text{Dense} \ l) \rangle$
shows $\langle \text{predict}_{\text{layer}_m} \ N \ (\text{Some} \ (\text{vs}::'a::\text{comm_ring Matrix.vec})) \ (\text{Dense} \ l)$
 $= \text{map_option} \ \text{vec_of_list} \ (\text{predict}_{\text{layer}_l} \ (\text{nn_matrix_to_list} \ N) \ (\text{Some} \ (\text{list_of_vec} \ \text{vs}))) \ (\text{layer_matrix_to_list} \ (\text{Dense} \ l)) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{Dense}_m \ \text{to}_l \ \text{eq}'$:

assumes $\langle \text{layer_consistent}_m \ N \ (\text{dim_vec} \ \text{vs}) \ (\text{Dense} \ l) \rangle$
shows $\langle \text{predict}_{\text{layer}_l} \ (\text{nn_matrix_to_list} \ N) \ (\text{Some} \ (\text{list_of_vec} \ \text{vs})) \ (\text{layer_matrix_to_list} \ (\text{Dense} \ l))$
 $= \text{map_option} \ \text{list_of_vec} \ (\text{predict}_{\text{layer}_m} \ N \ (\text{Some} \ (\text{vs}::'a::\text{comm_ring Matrix.vec})) \ (\text{Dense} \ l)) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{Activation}_m \ \text{to}_l \ \text{eq}$:

$\langle \text{predict}_{\text{layer}_m} \ N \ (\text{Some} \ \text{vs}) \ (\text{Activation} \ l)$
 $= \text{map_option} \ \text{vec_of_list} \ (\text{predict}_{\text{layer}_l} \ (\text{nn_matrix_to_list} \ N) \ (\text{Some} \ (\text{list_of_vec} \ \text{vs}))) \ (\text{layer_matrix_to_list} \ (\text{Activation} \ l)) \rangle$

l)))>
<proof>

lemma *Activation_m_to_l_eq'*:

<predict_{layer_l} (nn_matrix_to_list N) (Some (list_of_vec vs)) (layer_matrix_to_list (Activation l))
= map_option list_of_vec (predict_{layer_m} N (Some vs) (Activation l))>
<proof>

lemma *predict_seq_m_eq_l'*:

assumes <layers_consistent_m (|layers = l₀, activation_tab = activation_tab') (dim_vec (Inputs::'a::comm_ring Matrix.vec)) layers'>
and <valid_activation_tab_m activation_tab'>
shows <foldl (predict_{layer_m} (|layers = l₁, activation_tab = activation_tab')) (Some (Inputs)) (layers (|layers = layers', activation_tab = a₁)) =
map_option vec_of_list
(foldl (predict_{layer_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>

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>

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 (predict_{seq_layer_m} N Inputs) = predict_{seq_layer_l} (nn_matrix_to_list N) (list_of_vec Inputs)>
<proof>
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

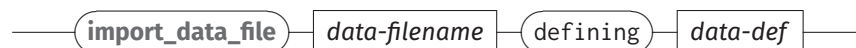
imports

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:



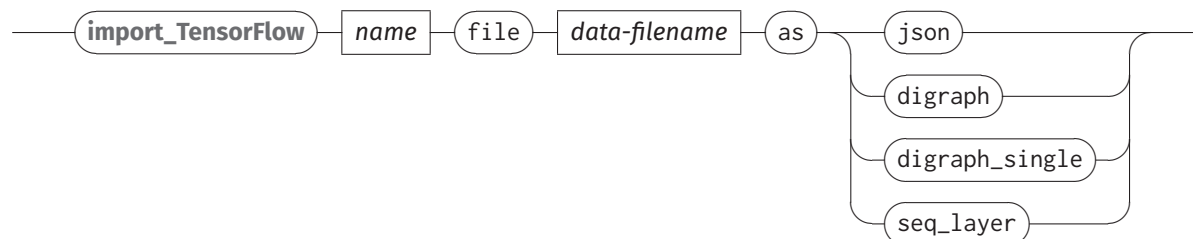
where *data-filename* is the path (file name) of the data file that should be read and *data-def* is the name the HOL definition representing the data is bound to. The data file should be a simple two-dimensional array of real numbers as, e.g., produced by NumPy's [11] `saveetxt` command:

```
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 $m_N9 \equiv Neuron\ (\varphi = Identity, \alpha = 1, \beta = 7082077 / 1000000000, uid = 9)$

definition $m_N10 \equiv Neuron\ (\varphi = Identity, \alpha = 1, \beta = 107544795 / 1000000000, uid = 10)$

definition $m_N11 \equiv Neuron\ (\varphi = Identity, \alpha = 1, \beta = -15743796 / 1000000000, uid = 11)$

definition $m_N12 \equiv Neuron\ (\varphi = Identity, \alpha = 1, \beta = 47920802 / 1000000000, uid = 12)$

definition $m_N13 \equiv Neuron\ (\varphi = Identity, \alpha = 1, \beta = -16364478 / 1000000000, uid = 13)$

definition $m_N14 \equiv Neuron\ (\varphi = Identity, \alpha = 1, \beta = -24132763 / 1000000000, uid = 14)$

definition $m_N15 \equiv Neuron\ (\varphi = Identity, \alpha = 1, \beta = -30579916 / 1000000000, uid = 15)$

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 $m_E12_16 \equiv (\omega = 1, tl = m_N12, hd = m_N16)$

definition $m_E13_17 \equiv (\omega = 1, tl = m_N13, hd = m_N17)$

definition $m_E14_18 \equiv (\omega = 1, tl = m_N14, hd = m_N18)$

definition $m_E15_19 \equiv (\omega = 1, tl = m_N15, hd = m_N19)$

definition $m_E9_12 \equiv (\omega = 8217673 / 200000000, tl = m_N9, hd = m_N12)$

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

```
⟨proof⟩
```

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!0)) E12_16))  
(Some (predictions!0!0))⟩  
value ⟨(checkget_result_singleton 0.15 (predictdigraph_single compass.NeuralNet (map_of_list (input!0)) E12_16))  
(Some (predictions!1!0))⟩  
value ⟨(checkget_result_singleton 0.15 (predictdigraph_single compass.NeuralNet (map_of_list (input!0)) E12_16))  
(Some (predictions!2!0))⟩  
value ⟨(checkget_result_singleton 0.15 (predictdigraph_single compass.NeuralNet (map_of_list (input!0)) E12_16))  
(Some (predictions!3!0))⟩
```

lemma compass_truth_table_predict:

```
⟨(predictdigraph_single compass.NeuralNet (map_of_list (input!0)) E12_16) ≈[0.0001] (Some (predictions!0!0))⟩  
⟨(predictdigraph_single compass.NeuralNet (map_of_list (input!1)) E12_16) ≈[0.0001] (Some (predictions!1!0))⟩  
⟨(predictdigraph_single compass.NeuralNet (map_of_list (input!2)) E12_16) ≈[0.0001] (Some (predictions!2!0))⟩
```

⟨(predict_{digraph_single} compass.NeuralNet (map_of_list (input!3)) E12_16) ≈[0.0001]≈_s (Some (predictions!3!0)))⟩
 ⟨proof⟩

lemma compass_truth_table_predict':

⟨(predict_{digraph_single_list} compass.NeuralNet (input!0) ≈[0.0001]≈_l (Some (predictions!0)))⟩
 ⟨(predict_{digraph_single_list} compass.NeuralNet (input!1) ≈[0.0001]≈_l (Some (predictions!1)))⟩
 ⟨(predict_{digraph_single_list} compass.NeuralNet (input!2) ≈[0.0001]≈_l (Some (predictions!2)))⟩
 ⟨(predict_{digraph_single_list} compass.NeuralNet (input!3) ≈[0.0001]≈_l (Some (predictions!3)))⟩
 ⟨proof⟩

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

⟨proof⟩

lemma Edges_equiv: compass.Edges = m_Edges

⟨proof⟩

lemma Graph_equiv: compass.Graph = m_Graph

⟨proof⟩

lemma φ_equiv: compass.φ_compass f = m_φ_compass f

⟨proof⟩

lemma NeuralNet_equiv: compass.NeuralNet = m_NeuralNet

⟨proof⟩

lemma < predict_{digraph_single_list} compass.NeuralNet = predict_{digraph_single_list} m_NeuralNet

⟨proof⟩

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 mIdentity    = 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 = mIdentity,  
   β = [9153944253921509 / 10000000000000000, - 959978699684143 / 10000000000000000, 7840137928724289 /  
100000000000000000],  
   ω = [[- 2865548133850977 / 10000000000000000, 1398887038230896 / 10000000000000000, - 4396021068096161 /  
100000000000000000,  
         - 3206970691680908 / 10000000000000000, - 25562143325805664 / 10000000000000000, 11852015256881714 /  
100000000000000000,  
         6039865016937256 / 10000000000000000, - 16825008392333984 / 10000000000000000, - 413370318710804 /  
100000000000000000],  
   [24456006288528442 / 10000000000000000, - 11522198915481567 / 10000000000000000, 4993317425251007 /  
100000000000000000,  
     - 17345187664031982 / 10000000000000000, 48335906863212585 / 10000000000000000, 1511125922203064 /  
100000000000000000,  
     - 36204618215560913 / 10000000000000000, 9508050084114075 / 10000000000000000, - 3617756962776184 /  
100000000000000000],  
   [704086497426033 / 10000000000000000, - 51195383071899414 / 10000000000000000, - 34204763174057007 /  
100000000000000000,  
     - 72454833984375 / 10000000000000000, - 33541640639305115 / 10000000000000000, 12738076448440552 /  
100000000000000000,  
     7601173520088196 / 10000000000000000, - 2638514041900635 / 10000000000000000, - 5478811264038086 /  
100000000000000000])))
```

```
definition m_dense_2 =
```

```
  Dense
```

```
  (|name = STR "dense_2", units = 4, φ = mIdentity,  
   β = [39810407906770706 / 10000000000000000, 874686986207962 / 10000000000000000, - 4944610595703125 /  
100000000000000000,  
     - 5116363242268562 / 10000000000000000],  
   ω = [[[ (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 ≡ 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** $m_φ_ran$: $\langle ran\ m_φ_compass = \{map\ identity\} \rangle$

$\langle proof \rangle$

interpretation $nn_{n.or}$: $neural_network_sequential_layers_l\ m_NeuralNet$

$\langle proof \rangle$

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 $\langle checkget_result_list\ 0.001\ (predict_{seq_layer_l}\ NeuralNet\ (input!0))\ (Some\ (predictions!0)) \rangle$

value $\langle checkget_result_list\ 0.001\ (predict_{seq_layer_l}\ NeuralNet\ (input!1))\ (Some\ (predictions!1)) \rangle$

value $\langle checkget_result_list\ 0.001\ (predict_{seq_layer_l}\ NeuralNet\ (input!2))\ (Some\ (predictions!2)) \rangle$

value $\langle checkget_result_list\ 0.001\ (predict_{seq_layer_l}\ NeuralNet\ (input!3))\ (Some\ (predictions!3)) \rangle$

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))>
<proof>

```

lemma $\langle 0.000001 \models_l \{input\} (predict_{seq_layer_l} NeuralNet) \{predictions\} \rangle \langle proof \rangle$

lemma *activation[simp]*: $\langle activation_tab NeuralNet = compass.\varphi_compass \rangle$
 $\langle proof \rangle$

lemma *layers[simp]*: $\langle layers NeuralNet = [dense_input, Layers.dense, dense_2, OUTPUT] \rangle$
 $\langle proof \rangle$

lemma *input[simp]*: $\langle in_deg_NN NeuralNet = 9 \rangle$
 $\langle proof \rangle$

import_data_file *model/compass.txt defining compass*

definition *classify_as* :: $\langle real\ list \Rightarrow nat \Rightarrow bool \rangle$ **where**
 $\langle classify_as\ xs\ n = (Option.bind (predict_{seq_layer_l} compass.NeuralNet\ xs) Prediction_Utils.pos_of_max = Some\ n) \rangle$

lemma *co[simp]*: $compass!0 = [1,1,1,$
 $1,1,0,$
 $1,0,1]$
 $\langle proof \rangle$

lemma *c1[simp]*: $compass!1 = [1,1,1,$
 $0,1,1,$
 $1,0,1]$
 $\langle proof \rangle$

lemma *c2[simp]*: $compass!2 = [1,0,1,$
 $0,1,1,$
 $1,1,1]$
 $\langle proof \rangle$

lemma *c3[simp]*: $compass!3 = [1,0,1,$
 $1,1,0,$
 $1,1,1]$
 $\langle proof \rangle$

lemma *classify_NW*: $\langle classify_as (compass!0) 0 \rangle$
 $\langle proof \rangle$

lemma *classify_NE*: $\langle classify_as (compass!1) 1 \rangle$
 $\langle proof \rangle$

lemma *classify_SE*: $\langle classify_as (compass!2) 2 \rangle$

```

⟨proof⟩
lemma classify_SW: ⟨classify_as (compass!3) 3⟩
⟨proof⟩

lemma compass_img_defined: ⟨((predict_seq_layer_l compass.NeuralNet xs) ≠ None) = (length xs = 9)⟩
⟨proof⟩

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 mldentity = 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 = mldentity,
    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 / 10000000000000, - 33541640639305115 / 10000000000000000, 12738076448440552 /
100000000000000000,
    7601173520088196 / 10000000000000000, - 2638514041900635 / 10000000000000000, - 5478811264038086 /
100000000000000000]]])

```

definition $m_dense_2 =$

```

Dense
(|name = STR "dense_2", units = 4,  $\varphi = m\_identity$ ,
  $\beta = \text{vec\_of\_list}$  [39810407906770706 / 100000000000000000, 874686986207962 / 10000000000000000, -
4944610595703125 / 100000000000000000,
 - 5116363242268562 / 100000000000000000],
  $\omega = \text{mat\_of\_cols\_list}$  3
 [[(9063153862953186 / 100000000000000000::real), - 142851984500885 / 10000000000000000, - 10823805332183838
 / 100000000000000000],
 [17654908895492554 / 100000000000000000, 1934271901845932 / 100000000000000000, 1214023232460022 /
100000000000000000],
 [- 17099318504333496 / 100000000000000000, - 7595149427652359 / 100000000000000000, - 12841564416885376
 / 100000000000000000],
 [- 615866482257843 / 100000000000000000, 1532884955406189 / 100000000000000000, 17860114574432373 /
100000000000000000]]])

```

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

$\langle m_NeuralNet = (|layers = m_Layers, activation_tab = m_ φ _compass) \rangle$

Locale Interpretations lemma $m_ φ _ran$: $\langle ran\ m_ φ _compass = \{map_vec\ identity\} \rangle$

$\langle proof \rangle$

interpretation nn_{nor} : $neural_network_sequential_layers_m\ m_NeuralNet$

$\langle proof \rangle$

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. $\varphi$ _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 <(predict_seq_layer_m compass.NeuralNet) (vec_of_list(input!0))>
```

```
value <checkget_result_matrix 0.001 (predict_seq_layer_m NeuralNet o the) (input!0) (predictions!0)>
```

```
value <checkget_result_matrix 0.001 (predict_seq_layer_m NeuralNet o the) (input!1) (predictions!1)>
```

```
value <checkget_result_matrix 0.001 (predict_seq_layer_m NeuralNet o the) (input!2) (predictions!2)>
```

```
value <checkget_result_matrix 0.001 (predict_seq_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 (predict_seq_layer_m NeuralNet ((vec_of_list (input!0)))))) ≈[0.001]≈I (Some (predictions!0))>
```

```
<(map_option list_of_vec (predict_seq_layer_m NeuralNet ((vec_of_list (input!1)))))) ≈[0.001]≈I (Some (predictions!1))>
```

```
<(map_option list_of_vec (predict_seq_layer_m NeuralNet ((vec_of_list (input!2)))))) ≈[0.001]≈I (Some (predictions!2))>
```

```
<(map_option list_of_vec (predict_seq_layer_m NeuralNet ((vec_of_list (input!3)))))) ≈[0.001]≈I (Some (predictions!3))>
```

```
<proof>
```

```
lemma <0.000001 ⊨I {input} (predict NeuralNet) {predictions}>
```

```
<proof>
```

```
lemma activation[simp]: <activation_tab NeuralNet = compass.φ_compass >
```

```
<proof>
```

```
lemma layers[simp]: <layers NeuralNet = [dense_input, Layers.dense, dense_2, OUTPUT] >
```

```
<proof>
```

```
lemma input[simp]: <in_deg_NN NeuralNet = 9 >
```

```
<proof>
```

```
import_data_file model/compass.txt defining compass
```

```
lemma co[simp]: compass!0 = [1,1,1,
```

```
1,1,0,
```

```
1,0,1]
```

```
<proof>
```

```
lemma c1[simp]: compass!1 = [1,1,1,
```

```
0,1,1,
```

```
1,0,1]
```

```
<proof>
```

```
lemma c2[simp]: compass!2 = [1,0,1,
```

```
0,1,1,
```

$1,1,1]$
 $\langle proof \rangle$

lemma $c3[simp]$: $compass!3 = [1,0,1,$
 $1,1,0,$
 $1,1,1]$
 $\langle proof \rangle$

lemma $compass_img_defined$: $\langle ((predict_{seq_layer_m} compass.NeuralNet xs) \neq None) = (length (list_of_vec xs) = 9) \rangle$
 $\langle proof \rangle$

definition $classify_as$:: $\langle real Matrix.vec \Rightarrow nat \Rightarrow bool \rangle$ **where**
 $\langle classify_as xs n = (Option.bind (predict_{seq_layer_m} compass.NeuralNet xs) pos_of_max = Some n) \rangle$

lemma $classify_NW$: $\langle classify_as (vec_of_list(compass!0)) 0 \rangle$
 $\langle proof \rangle$

lemma $classify_NE$: $\langle classify_as (vec_of_list(compass!1)) 1 \rangle$
 $\langle proof \rangle$

lemma $classify_SE$: $\langle classify_as (vec_of_list(compass!2)) 2 \rangle$
 $\langle proof \rangle$

lemma $classify_SW$: $\langle classify_as (vec_of_list(compass!3)) 3 \rangle$
 $\langle proof \rangle$

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].

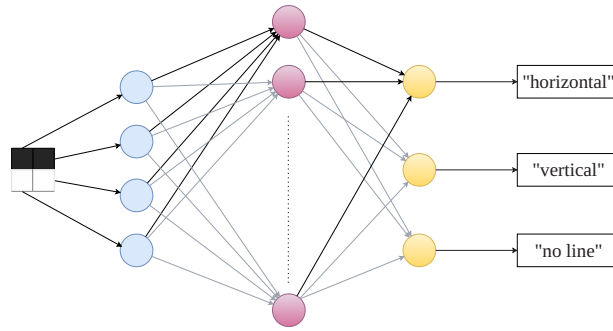


Figure 8.2: Neural network for classifying lines in 2×2 pixel grey scale images.

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, 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-1 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)))]
  )

```

$\langle _ \Rightarrow \text{None} \rangle$
 $\langle \text{proof} \rangle$

lemma grid_img_defined: $\langle (\text{predict}_{\text{seq_layer_l}} \text{grid.NeuralNet } xs) \neq \text{None} = (\text{length } xs = 4) \rangle$
 $\langle \text{proof} \rangle$

lemma grid_img_defined': $\langle (\exists y. (\text{predict}_{\text{seq_layer_l}} \text{grid.NeuralNet } xs) = \text{Some } y) = (\text{length } xs = 4) \rangle$
 $\langle \text{proof} \rangle$

lemma grid_image:
assumes $\langle (\text{predict}_{\text{seq_layer_l}} \text{grid.NeuralNet } xs) \neq \text{None} \rangle$
shows $\langle \text{the } (\text{predict}_{\text{seq_layer_l}} \text{grid.NeuralNet } xs) \in \{ [0, 0, 1], [0, 1, 0], [1, 0, 0] \} \rangle$
 $\langle \text{proof} \rangle$

lemma grid_image_approx:
 $\langle \text{ran } (\text{predict}_{\text{seq_layer_l}} \text{grid.NeuralNet}) \subseteq \{ [0, 0, 1], [0, 1, 0], [1, 0, 0] \} \rangle$
 $\langle \text{proof} \rangle$

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: $\langle \text{dom } (\text{predict}_{\text{seq_layer_l}} \text{grid.NeuralNet}) = \{ a. \text{length } a = 4 \} \rangle$
 $\langle \text{proof} \rangle$

definition range_of $x = (\text{if } x = (0::\text{real}) \text{ then } \{0..0.04::\text{real}\} \text{ else } \{0.96..1\})$

lemma $\langle x3 \in \{0.96..1.00\} \wedge x2 \in \{0.96..1.00\} \wedge x1 \in \{0.00..0.04\} \wedge x0 \in \{0.00..0.04\} \implies \text{predict}_{\text{seq_layer_l}} \text{grid.NeuralNet } [x3, x2, x1, x0] = \text{Some } [0, 1, 0] \rangle$
 $\langle \text{proof} \rangle$

lemma $\langle x3 \in \{0.00..0.04\} \wedge x2 \in \{0.00..0.04\} \wedge x1 \in \{0.96..1.00\} \wedge x0 \in \{0.96..1.00\} \implies \text{predict}_{\text{seq_layer_l}} \text{grid.NeuralNet } [x3, x2, x1, x0] = \text{Some } [0, 1, 0] \rangle$
 $\langle \text{proof} \rangle$

lemma $\langle x3 \in \{0.95..1.00\} \wedge x2 \in \{0.00..0.05\} \wedge x1 \in \{0.95..1.00\} \wedge x0 \in \{0.00..0.05\} \implies \text{predict}_{\text{seq_layer_l}} \text{grid.NeuralNet } [x3, x2, x1, x0] = \text{Some } [0, 0, 1] \rangle$
 $\langle \text{proof} \rangle$

lemma $\langle x3 \in \{0.00..0.1\} \wedge x2 \in \{0.96..1.00\} \wedge x1 \in \{0.00..0.1\} \wedge x0 \in \{0.96..1.00\} \implies \text{predict}_{\text{seq_layer_l}} \text{grid.NeuralNet } [x3, x2, x1, x0] = \text{Some } [0, 0, 1] \rangle$
 $\langle \text{proof} \rangle$

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:
 $\langle \models_{il} \{ \text{inputs} \} (\text{predict}_{\text{seq_layer_l}} \text{grid.NeuralNet}) \{ \text{intervals_of_l } 0.000001 \text{ predictions} \} \rangle$
 $\langle \text{proof} \rangle$

lemma grid_meets_expectations_max_classifier:
 $\langle \models_{il} \{ \text{inputs_small} \} (\text{predict}_{\text{seq_layer_l}} \text{grid.NeuralNet}) \{ \text{expectations_small} \} \rangle$
 $\langle \text{proof} \rangle$

lemma grid_min_delta_classifier:
 $\langle 1.0 \models \text{predict}_{\text{seq_layer_l}} \text{grid.NeuralNet} \rangle$

<proof>

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. 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*

Proving using the matrix prediction function.

lemma *grid_closed_mat* [*simp*]:
 $\langle \text{predict}_{\text{seq_layer_m}} \text{grid.NeuralNet} (\text{vec_of_list } xs) = (\text{case } xs \text{ of } [x_3, x_2, x_1, x_0] \Rightarrow \text{let } y = 2 * (x_3 + (x_2 * 2 + (x_1 * 4 + x_0 * 8)))$
in *Some* ($\text{vec_of_list}[(\text{if } y - 7 \leq 0 \text{ then } 0 \text{ else } 1) +$
 $((\text{if } y - 11 \leq 0 \text{ then } 0 \text{ else } 1) + (\text{if } y - 21 \leq 0 \text{ then } 0 \text{ else } 1)$
 $+ ((\text{if } y - 25 \leq 0 \text{ then } 0 \text{ else } 1) - (\text{if } y - 23 \leq 0 \text{ then } 0 \text{ else } 1))$
 $- (\text{if } y - 19 \leq 0 \text{ then } 0 \text{ else } 1)) - (\text{if } y - 9 \leq 0 \text{ then } 0 \text{ else } 1))$
 $- (\text{if } y - 5 \leq 0 \text{ then } 0 \text{ else } 1) + 1,$
 $(\text{if } y - 5 \leq 0 \text{ then } 0 \text{ else } 1) + ((\text{if } y - 23 \leq 0 \text{ then } 0 \text{ else } 1)$
 $- (\text{if } y - 25 \leq 0 \text{ then } 0 \text{ else } 1) - (\text{if } y - 7 \leq 0 \text{ then } 0 \text{ else } 1)),$
 $(\text{if } y - 9 \leq 0 \text{ then } 0 \text{ else } 1) + ((\text{if } y - 19 \leq 0 \text{ then } 0 \text{ else } 1)$
 $- (\text{if } y - 21 \leq 0 \text{ then } 0 \text{ else } 1) - (\text{if } y - 11 \leq 0 \text{ then } 0 \text{ else } 1))]$
 $\mid _ \Rightarrow \text{None}) \rangle$
 $\langle \text{proof} \rangle$

lemma *grid_img_defined_mat*: $\langle ((\text{predict}_{\text{seq_layer_m}} \text{grid.NeuralNet} (\text{vec_of_list } xs)) \neq \text{None}) = (\text{dim_vec} (\text{vec_of_list } xs) = 4) \rangle$
 $\langle \text{proof} \rangle$

lemma *grid_img_defined_mat'*: $\langle (\exists y. (\text{predict}_{\text{seq_layer_m}} \text{grid.NeuralNet} (\text{vec_of_list } xs)) = \text{Some} (\text{vec_of_list } y)) = (\text{dim_vec} (\text{vec_of_list } xs) = 4) \rangle$
 $\langle \text{proof} \rangle$

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 $\langle y \in \{ \text{vec_of_list } [0, 0, 1], \text{vec_of_list } [0, 1, 0], \text{vec_of_list } [1, 0, 0] \} \rangle$
 $\langle \text{proof} \rangle$

lemma *ran_aux*:
assumes $\langle \forall x. f x \neq \text{None} \longrightarrow \text{the } (f x) \in Y \rangle$
shows $\langle \text{ran } (f) \subseteq Y \rangle$
 $\langle \text{proof} \rangle$

lemma *grid_image_approx_mat*:
 $\langle \text{ran } (\lambda x. \text{predict}_{\text{seq_layer_m}} \text{grid.NeuralNet} (\text{vec_of_list } x))$
 $\subseteq \{ \text{vec_of_list } [0, 0, 1], \text{vec_of_list } [0, 1, 0], \text{vec_of_list } [1, 0, 0] \} \rangle$
 $\langle \text{proof} \rangle$

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*: $\langle \text{dom } (\lambda x. \text{predict}_{\text{seq_layer_m}} \text{grid.NeuralNet} (\text{vec_of_list } x)) = \{ a. \text{length } a = 4 \} \rangle$
 $\langle \text{proof} \rangle$

definition *range_of* $x = (\text{if } x = (0::\text{real}) \text{ then } \{0..0.04::\text{real}\} \text{ else } \{0.96..1\})$

lemma $\langle x3 \in \{0.96..1.00\} \wedge x2 \in \{0.96..1.00\} \wedge x1 \in \{0.00..0.04\} \wedge x0 \in \{0.00..0.04\} \implies \text{predict}_{\text{seq_layer_m}} \text{grid.NeuralNet} (\text{vec_of_list} [x3, x2, x1, x0]) = \text{Some}(\text{vec_of_list} [0, 1, 0]) \rangle$
<proof>

lemma $\langle x3 \in \{0.00..0.04\} \wedge x2 \in \{0.00..0.04\} \wedge x1 \in \{0.96..1.00\} \wedge x0 \in \{0.96..1.00\} \implies \text{predict}_{\text{seq_layer_m}} \text{grid.NeuralNet} (\text{vec_of_list} [x3, x2, x1, x0]) = \text{Some}(\text{vec_of_list} [0, 1, 0]) \rangle$
<proof>

lemma $\langle x3 \in \{0.95..1.00\} \wedge x2 \in \{0.00..0.05\} \wedge x1 \in \{0.95..1.00\} \wedge x0 \in \{0.00..0.05\} \implies \text{predict}_{\text{seq_layer_m}} \text{grid.NeuralNet} (\text{vec_of_list} [x3, x2, x1, x0]) = \text{Some}(\text{vec_of_list} [0, 0, 1]) \rangle$
<proof>

lemma $\langle x3 \in \{0.00..0.1\} \wedge x2 \in \{0.96..1.00\} \wedge x1 \in \{0.00..0.1\} \wedge x0 \in \{0.96..1.00\} \implies \text{predict}_{\text{seq_layer_m}} \text{grid.NeuralNet} (\text{vec_of_list} [x3, x2, x1, x0]) = \text{Some}(\text{vec_of_list} [0, 0, 1]) \rangle$
<proof>

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]:
 $\langle \text{predict}_{\text{seq_layer_m}}' \text{grid.NeuralNet} \text{ xs} = (\text{case } \text{xs} \text{ of } [x3, x2, x1, x0] \Rightarrow \text{let } y = 2 * (x3 + (x2 * 2 + (x1 * 4 + x0 * 8))) \text{ in } \text{Some} ([(if } y - 7 \leq 0 \text{ then } 0 \text{ else } 1) + ((if } y - 11 \leq 0 \text{ then } 0 \text{ else } 1) + ((if } y - 21 \leq 0 \text{ then } 0 \text{ else } 1) + ((if } y - 25 \leq 0 \text{ then } 0 \text{ else } 1) - (if } y - 23 \leq 0 \text{ then } 0 \text{ else } 1)) - (if } y - 19 \leq 0 \text{ then } 0 \text{ else } 1)) - (if } y - 9 \leq 0 \text{ then } 0 \text{ else } 1)) - (if } y - 5 \leq 0 \text{ then } 0 \text{ else } 1) + 1, (if } y - 5 \leq 0 \text{ then } 0 \text{ else } 1) + ((if } y - 23 \leq 0 \text{ then } 0 \text{ else } 1) - (if } y - 25 \leq 0 \text{ then } 0 \text{ else } 1) - (if } y - 7 \leq 0 \text{ then } 0 \text{ else } 1)), (if } y - 9 \leq 0 \text{ then } 0 \text{ else } 1) + ((if } y - 19 \leq 0 \text{ then } 0 \text{ else } 1) - (if } y - 21 \leq 0 \text{ then } 0 \text{ else } 1) - (if } y - 11 \leq 0 \text{ then } 0 \text{ else } 1))]) \mid _ \Rightarrow \text{None} \rangle$
<proof>

lemma *grid_img_defined'*: $\langle ((\text{predict}_{\text{seq_layer_m}}' \text{grid.NeuralNet} \text{ xs}) \neq \text{None}) = (\text{length } \text{xs} = 4) \rangle$
<proof>

lemma *grid_img_defined*: $\langle (\exists y. (\text{predict}_{\text{seq_layer_m}}' \text{grid.NeuralNet} \text{ xs}) = \text{Some } y) = (\text{length } \text{xs} = 4) \rangle$
<proof>

lemma *grid_image*:
assumes $\langle (\text{predict}_{\text{seq_layer_m}}' \text{grid.NeuralNet} \text{ xs}) \neq \text{None} \rangle$
shows $\langle \text{the } (\text{predict}_{\text{seq_layer_m}}' \text{grid.NeuralNet} \text{ xs}) \in \{ [0, 0, 1], [0, 1, 0] \} \rangle$

[1, 0, 0]⟩
 ⟨proof⟩

lemma grid_image_approx:

⟨ran (predict_{seq_layer_m} 'grid.NeuralNet) ⊆ {[0, 0, 1], [0, 1, 0], [1, 0, 0]}⟩
 ⟨proof⟩

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}⟩
 ⟨proof⟩

lemma grid_dom_mat': ⟨dom (λ x . predict_{seq_layer_m} grid.NeuralNet (vec_of_list x)) = {a. length a = 4}⟩
 ⟨proof⟩

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]⟩
 ⟨proof⟩

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]⟩
 ⟨proof⟩

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]⟩
 ⟨proof⟩

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]⟩
 ⟨proof⟩

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}⟩
 ⟨proof⟩

lemma grid_meets_expectations_max_classifier:

⟨|=_l {inputs_small} (predict_{seq_layer_m} 'grid.NeuralNet) {expectations_small}⟩
 ⟨proof⟩

lemma grid_min_delta_classifier:

⟨1.0 |= predict_{seq_layer_m} 'grid.NeuralNet⟩
 ⟨proof⟩

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.