Formalizing Neural Networks

Achim D. Brucker Amy Stell

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

February 6, 2026

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 Activion 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) combiners 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) combiners 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 verfication 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.

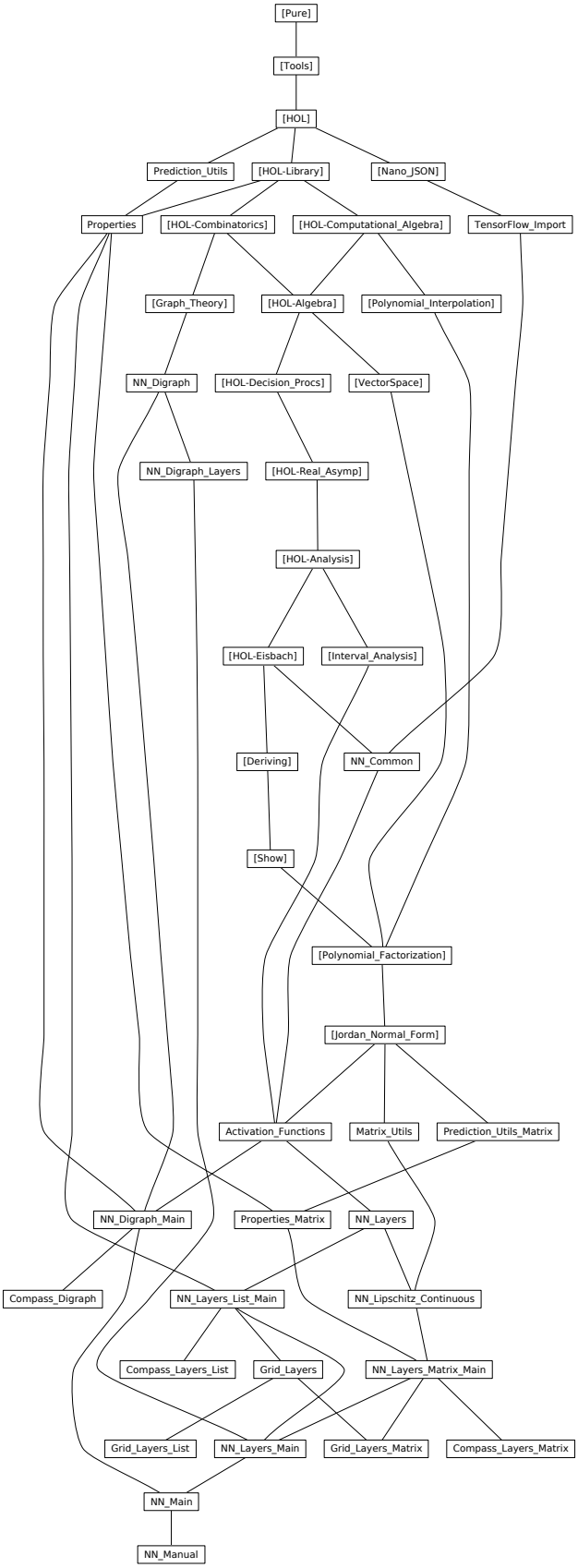


Figure 1.1: The Dependency Graph of the Isabelle Theories.

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* ⟹ (*map2* (∗) *xs* (*ys*)) = *map* (λ *i*. *xs*!*i* ∗ *ys*!*i*) [*o*..< *length xs*]›
⟨*proof*⟩

lemma *map2_to_map_idx*:
‹(*map2* (∗) *xs* (*ys*)) = *map* (λ *i*. *xs*!*i* ∗ *ys*!*i*) [*o*..< *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* ⇒ '*a* :: *semiring_0 Matrix.mat* ⇒ '*a Matrix.vec* (**infixl** $_v*$ 70)
where *v* $_v*$ *A* ≡ *vec* (*dim_col A*) (λ *i*. *col A i* ∙ *v*)

lemma *dim_mult_vec_mat*: ‹*dim_vec* (*v* $_v*$ *A*) = *dim_col A*›
⟨*proof*⟩

lemma *index_mult_vec_mat*: ‹ *i* < *dim_col A* ⟹ (*v* $_v*$ *A*) $ *i* = *col A i* ∙ *v*›
⟨*proof*⟩

lemma *dim_col_mat_list*: ‹∀ *m* ∈ *set* (*mat_to_list M*). *dim_col M* = *length m* ›
⟨*proof*⟩

lemma *dim_col_mat_list'*: ‹*mat_to_list M* ≠ [] ⟹ *dim_col M* = *length* (*hd* (*mat_to_list M*))›
⟨*proof*⟩

lemma *scalar_prod_list*:
‹((*vec_of_list v*) ∙ (*vec_of_list w*)) = (∑ *i* ∈ {*o* ..< *length w*}. *v*!*i* ∗ *w*!*i*)›

⟨proof⟩

lemma *dim_col_mat_of_col_list*: ‹*dim_col* (*mat_of_cols_list n As*) = *length As*›
⟨proof⟩

lemma *dim_row_mat_of_col_list*: ‹*dim_row* (*mat_of_cols_list n As*) = *n*›
⟨proof⟩

lemma *dim_col_mat_of_row_list*: ‹*dim_col* (*mat_of_rows_list n As*) = *n*›
⟨proof⟩

lemma *dim_row_mat_of_row_list*: ‹*dim_row* (*mat_of_rows_list n As*) = *length As*›
⟨proof⟩

lemma *vec_of_list_ext*: ‹*vec_of_list xs* = *vec_of_list ys* ⟹ *xs* = *ys*›
⟨proof⟩

lemma *list_of_vec_ext*: ‹*list_of_vec xs* = *list_of_vec ys* ⟹ *xs* = *ys*›
⟨proof⟩

lemma *map_if_lam*:
‹*map* (λ *i*. *if i* < *n then P*(*i*) *else Q*(*i*)) [*o*..<*n*] = *map* (λ *i*. *P*(*i*)) [*o*..<*n*]›
⟨proof⟩

lemma *map_if_lam′*:
‹*map* (λ *i*. *if p* ∧ *i* < *n then* (*P i*) *else* (*Q i*)) [*o*..<*n*] = *map* (λ *i*. *if p then* (*P i*) *else* (*Q i*)) [*o*..<*n*]›
⟨proof⟩

lemma *map_if_lam′′*:
‹*map* (λ*i*. *map* (λ*ia*. *if i* < *n then P i ia else Q i ia*) [*o*..<*m*]) [*o*..<*n*]
=*map* (λ*i*. *map* (λ*ia*. *P i ia*) [*o*..<*m*]) [*o*..<*n*]›
⟨proof⟩

lemma *vec_add_list*:
assumes ‹*length v* = *length w*›
shows ‹*list_of_vec* ((*vec_of_list v*) + (*vec_of_list w*)) = *map2* (+) *v w*›
⟨proof⟩

lemma *vec_add_list′*:
assumes ‹*length v* = *length w*›
shows ‹((*vec_of_list v*) + (*vec_of_list w*)) = *vec_of_list* (*map2* (+) *v w*)›
⟨proof⟩

lemma *mat_col_list*:
assumes ‹*i* < *length As*›
and ‹∀ *a* ∈ *set As*. ∀ *a′* ∈ *set As*. *length a* = *length a′* ∧ *a* ≠ []›
and ‹*d* = *length* (*hd As*)›
shows ‹*list_of_vec* (*col* (*mat_of_cols_list d As*) *i*) = *As*!*i*›
⟨proof⟩

lemma *mult_vec_mat_col_list*:
assumes ‹*length vs*=*n*›
and ‹∀ *a* ∈ *set As*. ∀ *a′* ∈ *set As*. *length a* = *length a′* ∧ *a* ≠ []›
and ‹*length* (*hd As*)=*d*›

and ‹*length As*=*n*›
and ‹*As* ≠ []›
shows ‹*list_of_vec* ((*vec_of_list vs*) $_v*$ (*mat_of_cols_list d As*)) = *map* (λ*i*. $\sum$ *ia* = *o*..<*length vs*. *As* ! *i* ! *ia* ∗ *vs* ! *ia*) [*o*..<*n*] ›
⟨*proof*⟩

lemma *mult_vec_mat_row_list*:
assumes ‹*length vs*=*d*›
and ‹∀ *a* ∈ *set As*. ∀ *a'* ∈ *set As*. *length a* = *length a'* ∧ *a* ≠ []›
and ‹*length* (*hd As*)=*d*›
and ‹*length As*=*n*›
and ‹*As* ≠ []›
shows ‹*list_of_vec* ((*vec_of_list vs*) $_v*$ (*mat_of_rows_list d As*)) = *map* (λ*i*. $\sum$ *ia* = *o*..<*length vs*. *map* (λ*ia*. *As* ! *ia* ! *i*) [*o*..<*length As*] ! *ia* ∗ *vs* ! *ia*) [*o*..<*d*]›
⟨*proof*⟩

lemma *mult_vec_mat_row_list'*:
assumes ‹*length vs*=*d*›
and ‹∀ *a* ∈ *set As*. ∀ *a'* ∈ *set As*. *length a* = *length a'* ∧ *a* ≠ []›
and ‹*length* (*hd As*)=*d*›
and ‹*length As*=*n*›
and ‹*As* ≠ []›
shows ‹((*vec_of_list vs*) $_v*$ (*mat_of_rows_list d As*)) = *vec_of_list* (*map* (λ*i*. $\sum$ *ia* = *o*..<*length vs*. *map* (λ*ia*. *As* ! *ia* ! *i*) [*o*..<*length As*] ! *ia* ∗ *vs* ! *ia*) [*o*..<*d*])›
⟨*proof*⟩

lemma *col_of_rows_list*:
assumes ‹*d* = *Min* (*set* (*map length As*))›
and ‹*i* < *d*›
shows ‹*list_of_vec* (*col* (*mat_of_rows_list d As*) *i*) = *map* (λ *as*. (*as*!*i*)) *As*›
⟨*proof*⟩

lemma *col_of_rows_list'*:
assumes ‹∀ *as* ∈ *set As*. *length as* = *d*›
and ‹*As* ≠ []›
shows ‹(*col* (*mat_of_rows_list d As*) *i*) = *vec_of_list* (*map* (λ *as*. (*as*!*i*)) *As*)›
⟨*proof*⟩

lemma *list_mat*: ‹*mat_of_rows_list* (*dim_col A*) (*mat_to_list A*) = *A*›
⟨*proof*⟩

lemma *list_mat_transpose_transpose*: ‹(*mat_of_rows_list* (*dim_col* x^T) (*mat_to_list* x^T))^T = *x*›
⟨*proof*⟩

lemma *mat_list*:
‹∀ *r* ∈ *set*(*rs*). *length r* = *dimc* ⟹*mat_to_list* (*mat_of_rows_list dimc rs*) = *rs*›
⟨*proof*⟩

lemma *dim_row_list*: ‹*dim_row m* = *length* (*mat_to_list m*)›
⟨*proof*⟩

lemma *dim_col_list*: ‹∀ *c* ∈ *set* (*mat_to_list m*). *length c* = *dim_col m*›
⟨*proof*⟩

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.

$\langle ML \rangle$

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

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

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

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

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

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

Instead of storing the biases and weights in the JSON file, the exported JSON only contains the type information (here: `float32`) refers to an external file (here: `group1-shard1of1.bin` that stores the actual value. In our example, this external file has a size of 12 bytes, storing three 32 Bit floating point numbers (encoding as IEEE floating point using a Little Endian encoding. The order of the biases and weights corresponds to the order and shape of their references in the original JSON file. In our example, the function `TensorFlow_Json.transform_json` results in the following transformed `weightsManifest`:

```json
"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 a 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 / 100000000000000000),
                  NUMBER
                   (− 2398796558380127 / 1000000000000000),
                  NUMBER
                   (− 46464818716049194 / 100000000000000000),
                  NUMBER (1946548342704773 / 1000000000000000)],
                 ARRAY
                 [NUMBER
                   (14860405027866364 / 100000000000000000),
                  NUMBER
                   (− 7789374142885208 / 100000000000000000),
                  NUMBER
                   (10928256511688232 / 100000000000000000),
                  NUMBER
                   (− 952406108379364 / 1000000000000000)],
                 ARRAY
                 [NUMBER
                   (24455930292606354 / 100000000000000000),
                  NUMBER (5169432163238525 / 100000000000000000),
                  NUMBER
                   (− 14084954261779785 / 100000000000000000),
                  NUMBER
                   (− 6348744630813599 /
                    100000000000000000)]])])]),
           (''bias_initializer'',
            OBJECT
             [(''class_name'', STRING ''Constant''),
              (''config'',
               OBJECT
                [(''value'',
                  ARRAY
                   [NUMBER (4792080223560333 / 100000000000000000),
                    NUMBER
                     (− 16364477574825287 / 100000000000000000),
                    NUMBER
                     (− 24132762849330902 / 100000000000000000),
                    NUMBER
                     (− 3057991564273834 / 100000000000000000)])])]),
           (''kernel_regularizer'', NULL),
           (''bias_regularizer'', NULL),
           (''activity_regularizer'', NULL),
           (''kernel_constraint'', NULL),
           (''bias_constraint'', NULL)])])])]),
  (''training_config'',
   OBJECT
    [(''loss'', STRING ''categorical_crossentropy''),
     (''metrics'',
      ARRAY
       [ARRAY
        [OBJECT
         [(''class_name'', STRING ''MeanMetricWrapper''),
          (''config'',
```

```
            OBJECT
             [(''name'', STRING ''binary_accuracy''),
              (''dtype'', STRING ''float32''),
              (''fn'', STRING ''binary_accuracy'')])]]]),
     (''weighted_metrics'', NULL), (''loss_weights'', NULL),
     (''optimizer_config'',
      OBJECT
       [(''class_name'', STRING ''Adam''),
        (''config'',
         OBJECT
          [(''name'', STRING ''Adam''),
           (''learning_rate'', NUMBER (1 / 1000)), (''decay'', NUMBER 0),
           (''beta_1'', NUMBER (9 / 10)),
           (''beta_2'', NUMBER (999 / 1000)),
           (''epsilon'', NUMBER (1 / 10000000)),
           (''amsgrad'', BOOL False)])])])]),
(''weightsManifest'',
 ARRAY
  [OBJECT
    [(''name'', STRING ''dense/kernel''),
     (''shape'',
      ARRAY
       [ARRAY
         [NUMBER (6684626 / 10000000), NUMBER (628606 / 10000000),
          NUMBER (9863281 / 100000000)],
        ARRAY
         [NUMBER (− 12952799 / 10000000), NUMBER (3662836 / 10000000),
          NUMBER (9530481 / 10000000)],
        ARRAY
         [NUMBER (− 2857958 / 10000000), NUMBER (6922799 / 10000000),
          NUMBER (35006753 / 100000000)],
        ARRAY
         [NUMBER (17300206 / 10000000), NUMBER (− 37598428 / 100000000),
          NUMBER (7897923 / 10000000)],
        ARRAY
         [NUMBER (63918763 / 100000000), NUMBER (15055849 / 100000000),
          NUMBER (− 58135855 / 100000000)],
        ARRAY
         [NUMBER (− 13919318 / 10000000), NUMBER (10981513 / 10000000),
          NUMBER (5679722 / 10000000)],
        ARRAY
         [NUMBER (− 45270395 / 100000000), NUMBER (17104555 / 1000000000),
          NUMBER (5311743 / 10000000)],
        ARRAY
         [NUMBER (13654941 / 10000000), NUMBER (7420693 / 10000000),
          NUMBER (− 9090567 / 10000000)],
        ARRAY
         [NUMBER (− 18450487 / 100000000), NUMBER (15639223 / 100000000),
          NUMBER (− 4547925 / 10000000)]])],
   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 / 10000000),
         NUMBER (− 4646482 / 10000000), NUMBER (19465483 / 10000000)],
       ARRAY
        [NUMBER (14860405 / 100000000), NUMBER (− 7789374 / 100000000),
         NUMBER (10928257 / 10000000), NUMBER (− 9524061 / 10000000)],
       ARRAY
        [NUMBER (2445593 / 10000000), NUMBER (5169432 / 10000000),
         NUMBER (− 14084954 / 10000000),
         NUMBER (− 63487446 / 100000000)]])],
  OBJECT
   [(''name'', STRING ''dense_2/bias''),
    (''shape'',
     ARRAY
      [ARRAY
        [NUMBER (47920802 / 100000000), NUMBER (− 16364478 / 100000000),
         NUMBER (− 24132763 / 100000000),
         NUMBER (− 30579916 / 100000000)]])])]
```

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 0*)›

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
‹$exp_{taylor}\ n\ x = (\sum i = 0..n\ .\ x\hat{}i\ /\ fact\ i)$›

lemma $exp_{taylor2}$: $exp_{taylor}\ 2\ (x::real) = (1::real) + x + x\hat{}2/2$
⟨*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*} ⇒ '*a*› **where**
‹$binary\_step = (\lambda\ v.\ if\ v \leq 0\ then\ 0\ else\ 1)$›

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 *sigmoid* :: *real* ⇒ *real* **where**
‹*sigmoid* = (λ*v*. 1 / (1 + *exp*(−*v*)))›

definition
‹*sigmoid*$_{taylor}$ *n* = (λ*v*. 1 / (1 + (*exp*$_{taylor}$ *n* (−*v*))))›

lemma ‹*sigmoid* = (*logistic* (1.0::*real*) 1.0 0)›
⟨*proof*⟩
lemma ‹*sigmoid*$_{taylor}$ *n* = (*logistic*$_{taylor}$ *n* (1.0::*real*) 1.0 0)›
⟨*proof*⟩

definition
‹*swish* = (λ*v*. *v* ∗ (*sigmoid v*))›
definition
‹*swish*$_{taylor}$ *n* = (λ*v*. *v* ∗ (*sigmoid*$_{taylor}$ *n v*))›
definition
‹*relu* = (λ*v*. *max* 0 *v*)›

definition
‹*generalized_relu* α *m t* = (λ*v*. *case m of Some m*′ ⇒ *min* (*if v* ≤ *t then* α ∗ *v else v*) *m*′
| *None* ⇒ *if v* ≤ *t then* α ∗ *v else v*)›

lemma ‹*relu* = (*generalized_relu* (0.0::*real*) *None* (0.0))›
⟨*proof*⟩

definition
‹*softplus* = (λ*v*. *ln* (1 + *exp v*))›

definition
‹*elu* α = (λ*v*. *if v* ≤ 0 *then* α ∗ ((*exp v*)−1) *else v*)›
definition
‹*elu*$_{taylor}$ *n* α = (λ*v*. *if v* ≤ 0 *then* α ∗ ((*exp*$_{taylor}$ *n v*)−1) *else v*)›
definition
‹*selu* = (λ*v*. *let* α = 1.67326324;
scale = 1.05070098
in if v ≤ 0 *then scale* ∗ α ∗ ((*exp v*)−1) *else scale* ∗ *v*)›
definition
‹*selu*$_{taylor}$ *n* = (λ*v*. *let* α = 1.67326324;
scale = 1.05070098
in if v ≤ 0 *then scale* ∗ α ∗ ((*exp*$_{taylor}$ *n v*)−1) *else scale* ∗ *v*)›

definition
‹*prelu* α = (λ*v*. *if v* < 0 *then* α ∗ *v else v*)›
definition
‹*silu* = (λ*v*. *v* / (1 + (*exp* (−*v*))))›
definition
‹*silu*$_{taylor}$ *n* = (λ*v*. *v* / (1 + (*exp*$_{taylor}$ *n* (−*v*))))›
definition
‹*gaussian* = (λ*v*. *exp* (− *v*^{2}))›
definition
‹*gaussian*$_{taylor}$ *n* = (λ*v*. *exp*$_{taylor}$ *n* (− *v*^{2}))›

definition
‹*hard_sigmoid* = $(\lambda v.$ *if* $v < -2.5$ *then* 0 *else if* $v > 2.5$ *then* 1 *else* $0.2 * v + 0.5)$›
definition
‹*gelu_approx* = $(\lambda v.$ $0.5 * v * (1 +$ *tanh*$($*sqrt*$(2 /$ *pi*$) * (v + 0.044715 * v * v * v))))$›

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

definition *softmax* :: $('a::\{$*banach*,*real_normed_algebra_1*,*inverse*$\})$ *list* $\Rightarrow$ *'a list* **where**
‹*softmax vs* = *map* $(\lambda$ *v*. *exp v* / $(\sum v' \leftarrow$*vs*. *exp v'*$))$ *vs*›

definition *msoftmax* :: $('a::\{$*banach*,*real_normed_algebra_1*,*inverse*$\})$ *vec* $\Rightarrow$ *'a vec* **where**
‹*msoftmax vs* = *map_vec* $(\lambda$ *v*. *exp v* / $(\sum v' \leftarrow$ (*list_of_vec vs*). *exp v'*$))$ *vs*›

definition $softmax_{taylor}$ :: *nat* $\Rightarrow$ $('a::\{$*banach*,*real_normed_algebra_1*,*inverse*$\})$ *list* $\Rightarrow$ *'a list* **where**
‹$softmax_{taylor}$ *n vs* = *map* $(\lambda$ *v*. $(exp_{taylor}$ *n v*$)$ / $(\sum v' \leftarrow$*vs*. $(exp_{taylor}$ *n v'*$)))$ *vs*›

definition $msoftmax_{taylor}$ :: *nat* $\Rightarrow$ $('a::\{$*banach*,*real_normed_algebra_1*,*inverse*$\})$ *vec* $\Rightarrow$ *'a vec* **where**
‹$msoftmax_{taylor}$ *n vs* = *map_vec* $(\lambda$ *v*. $(exp_{taylor}$ *n v*$)$ / $(\sum v' \leftarrow$ (*list_of_vec vs*). $(exp_{taylor}$ *n v'*$)))$ *vs*›

lemma $softmax_{taylor2}$:
$softmax_{taylor}$ *2 vs* = *map* $(\lambda$ (*v*::*real*). $(1 + v + v^2/2)$ / (*foldl* $(+)$ *0* (*map* $(\lambda$ *v'*. $1 + v' + v'^2/2)$ *vs*))) *vs*
⟨*proof*⟩

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

definition
‹*argmax vs* = *map* $(\lambda$ *v*. *if v* = *Max* (*set vs*) *then 1 else 0*$)$ *vs*›

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

3.1.2 Derivatives of Activation Functions

lemma *has_real_derivative_transform*:
‹$x \in s \Longrightarrow (\bigwedge x.\ x \in s \Longrightarrow g\ x = f\ x) \Longrightarrow (f\ \text{has\_real\_derivative}\ f')\ (\text{at}\ x\ \text{within}\ s)$
$\Longrightarrow (g\ \text{has\_real\_derivative}\ f')\ (\text{at}\ x\ \text{within}\ s)$›
⟨*proof*⟩

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

definition ‹$\text{identity}' \quad = (\lambda v.\ 1.0)$›
lemma *identity'*[*simp*]: ‹$(\text{identity}\ \text{has\_real\_derivative}\ (\text{identity}'\ v))\ (\text{at}\ v)$›
⟨*proof*⟩

definition ‹$\text{logistic}'\ L\ k\ v_0 = (\lambda v.\ (\exp(\ (-k)*(v-v_0)) * k * L)\ /\ (1 + \exp((-k)*(v-v_0)))^2)$›
lemma *logistic'*[*simp*]: ‹$((\text{logistic}\ L\ k\ v_0)\ \text{has\_real\_derivative}\ ((\text{logistic}'\ L\ k\ v_0)\ v))\ (\text{at}\ v)$›
⟨*proof*⟩

definition ‹$\tanh' \quad = (\lambda v.\ 1 - ((\tanh v)^2))$›
lemma *tanh'*[*simp*]: ‹$(\tanh\ \text{has\_real\_derivative}\ (\tanh'\ v))\ (\text{at}\ v)$›
⟨*proof*⟩

definition ‹$\text{softplus}' \quad = (\lambda v.\ 1\ /\ (1 + \exp(-v)))$›
lemma *softplus'*[*simp*]: ‹$(\text{softplus}\ \text{has\_real\_derivative}\ (\text{softplus}'\ v))\ (\text{at}\ v)$›
⟨*proof*⟩

definition ‹$\text{prelu1}' \quad = (\lambda v.\ 1)$›
lemma *prelu1'*[*simp*]: ‹$((\text{prelu}\ 1)\ \text{has\_real\_derivative}\ (\text{prelu1}'\ v))\ (\text{at}\ v)$›
⟨*proof*⟩

definition ‹$\text{silu}' \quad = (\lambda v.\ (1 + \exp(-v) + v * (\exp(-v))\ )\ /\ ((1 + \exp(-v))^2))$›
lemma *silu'*[*simp*]: ‹$(\text{silu}\ \text{has\_real\_derivative}\ (\text{silu}'\ v))\ (\text{at}\ v)$›
⟨*proof*⟩

definition ‹$\text{gaussian}' \quad = (\lambda v.\ -2 * v * \exp(-v^2))$›
lemma *gaussian'*[*simp*]: ‹$(\text{gaussian}\ \text{has\_real\_derivative}\ (\text{gaussian}'\ v))\ (\text{at}\ v)$›
⟨*proof*⟩

3.1.3 Single Class Folding Activation Functions

datatype $\text{activation}_{single} = \text{Identity} \mid \text{Sign} \mid \text{BinaryStep} \mid \text{Logistic real real real} \mid \text{Logistic}_{taylor}\ \text{nat real real real}$
$\mid \text{Tanh} \mid \text{Sigmoid} \mid \text{Sigmoid}_{taylor}\ \text{nat} \mid \text{ReLU} \mid \text{GReLU real ‹real option› real}$
$\mid \text{Softplus} \mid \text{SoftSign} \mid \text{Swish} \mid \text{Swish}_{taylor}\ \text{nat} \mid \text{GeLUapprox} \mid \text{ELU real}$
$\mid \text{ELU}_{taylor}\ \text{nat real} \mid \text{SELU} \mid \text{SELU}_{taylor}\ \text{nat} \mid \text{PReLU real} \mid \text{SiLU} \mid \text{SiLU}_{taylor}\ \text{nat}$
$\mid \text{Gaussian} \mid \text{Gaussian}_{taylor}\ \text{nat} \mid \text{Exp} \mid \text{Exp}_{taylor}\ \text{nat} \mid \text{HardSigmoid}$

fun φ_{single}:: ‹$\text{activation}_{single} \Rightarrow (\text{real} \Rightarrow \text{real})\ \text{option}$› **where**
‹$\varphi_{single}\ \text{Identity} \quad = \text{Some identity}$›
| ‹$\varphi_{single}\ \text{Sign} \quad = \text{Some sign}$›
| ‹$\varphi_{single}\ \text{BinaryStep} \quad = \text{Some binary\_step}$›
| ‹$\varphi_{single}\ \text{SoftSign} \quad = \text{Some softsign}$›
| ‹$\varphi_{single}\ (\text{Logistic}\ L\ k\ v_0) = \text{Some}\ (\text{logistic}\ L\ k\ v_0)$›
| ‹$\varphi_{single}\ (\text{Logistic}_{taylor}\ n\ L\ k\ v_0) = \text{Some}\ (\text{logistic}_{taylor}\ n\ L\ k\ v_0)$›
| ‹$\varphi_{single}\ \text{Sigmoid} \quad = \text{Some sigmoid}$›

| ‹φ_{single} (*Sigmoid*$_{taylor}$ *n*) = *Some* (*sigmoid*$_{taylor}$ *n*)›
| ‹φ_{single} *Swish* = *Some swish*›
| ‹φ_{single} (*Swish*$_{taylor}$ *n*) = *Some* (*swish*$_{taylor}$ *n*)›
| ‹φ_{single} *Tanh* = *Some tanh*›
| ‹φ_{single} *ReLU* = *Some relu*›
| ‹φ_{single} *GeLUapprox* = *Some gelu_approx*›
| ‹φ_{single} (*GReLU* α *m t*) = *Some* (*generalized_relu* α *m t*)›
| ‹φ_{single} *Softplus* = *Some softplus*›
| ‹φ_{single} (*ELU* α) = *Some* (*elu* α)›
| ‹φ_{single} (*ELU*$_{taylor}$ *n* α) = *Some* (*elu*$_{taylor}$ *n* α)›
| ‹φ_{single} *SELU* = *Some selu*›
| ‹φ_{single} (*SELU*$_{taylor}$ *n*) = *Some* (*selu*$_{taylor}$ *n*)›
| ‹φ_{single} *Exp* = *Some exp*›
| ‹φ_{single} (*Exp*$_{taylor}$ *n*) = *Some* (*exp*$_{taylor}$ *n*)›
| ‹φ_{single} *HardSigmoid* = *Some hard_sigmoid*›
| ‹φ_{single} (*PReLU* α) = *Some* (*prelu* α)›
| ‹φ_{single} *SiLU* = *Some silu*›
| ‹φ_{single} (*SiLU*$_{taylor}$ *n*) = *Some* (*silu*$_{taylor}$ *n*)›
| ‹φ_{single} *Gaussian* = *Some gaussian*›
| ‹φ_{single} (*Gaussian*$_{taylor}$ *n*) = *Some* (*gaussian*$_{taylor}$ *n*)›

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

fun φ_{single}':: ‹*activation*$_{single}$ ⇒ (*real* ⇒ *real option*)› **where**
‹φ_{single}' *Identity* = (λ*v*. *Some* (*identity' v*))›
| ‹φ_{single}' *Sign* = (λ*v*. *None*)›
| ‹φ_{single}' *BinaryStep* = (λ*v*. *None*)›
| ‹φ_{single}' (*Logistic L k v*$_0$) = (λ*v*. *Some* (*logistic' L k v*$_0$ *v*))›
| ‹φ_{single}' (*Logistic*$_{taylor}$ *n L k v*$_0$) = (λ*v*. *None*)›
| ‹φ_{single}' *Tanh* = (λ*v*. *Some* (*tanh' v*))›
| ‹φ_{single}' *ReLU* = (λ*v*. *None*)›
| ‹φ_{single}' *Softplus* = (λ*v*. *Some* (*softplus' v*))›
| ‹φ_{single}' (*ELU* α) = (λ*v*. *None*)›
| ‹φ_{single}' (*ELU*$_{taylor}$ *n* α) = (λ*v*. *None*)›
| ‹φ_{single}' (*PReLU* α) = (λ *v*. *if* α = *1 then Some* (*prelu1' v*) *else None*)›
| ‹φ_{single}' *SiLU* = (λ*v*. *Some* (*silu' v*))›
| ‹φ_{single}' (*SiLU*$_{taylor}$ *n*) = (λ*v*. *None*)›
| ‹φ_{single}' *Gaussian* = (λ*v*. *Some* (*gaussian' v*))›
| ‹φ_{single}' (*Gaussian*$_{taylor}$ *n*) = (λ*v*. *None*)›
| ‹φ_{single}' (*GReLU v va vb*) = (λ*v*. *None*)›
| ‹φ_{single}' *GeLUapprox* = (λ*v*. *None*)›
| ‹φ_{single}' *Sigmoid* = (λ*v*. *None*)›
| ‹φ_{single}' (*Sigmoid*$_{taylor}$ *n*) = (λ*v*. *None*)›
| ‹φ_{single}' *SoftSign* = (λ*v*. *None*)›
| ‹φ_{single}' *Swish* = (λ*v*. *None*)›
| ‹φ_{single}' (*Swish*$_{taylor}$ *n*) = (λ*v*. *None*)›
| ‹φ_{single}' *SELU* = (λ*v*. *None*)›
| ‹φ_{single}' (*SELU*$_{taylor}$ *n*) = (λ*v*. *None*)›
| ‹φ_{single}' *Exp* = (λ *v*. *Some* (*exp v*))›
| ‹φ_{single}' (*Exp*$_{taylor}$ *n*) = (λ *v*. *None*)›
| ‹φ_{single}' *HardSigmoid* = (λ*v*. *None*)›

The function φ_{single}' 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 ‹$v \in (dom\ (\varphi_{single}{}'\ a))$›
shows ‹$((\lambda\ v.\ the\ (\varphi_{single}\ a)\ v)\ has\_real\_derivative\ (the\ (\varphi_{single}{}'\ a\ v)))\ (at\ v\ within\ (dom\ (\varphi_{single}{}'\ a)))$›
$\langle proof \rangle$

3.1.4 Multiclass Folding Activation Functions

datatype $activation_{multi} = mIdentity \mid mSign \mid mBinaryStep \mid mLogistic\ real\ real\ real \mid mLogistic_{taylor}\ nat\ real\ real\ real$
$\mid mTanh \mid mSigmoid \mid mSigmoid_{taylor}\ nat \mid mReLU \mid mGReLU\ real$ ‹real option› $real$
$\mid mSoftplus \mid mSoftSign \mid mSwish \mid mSwish_{taylor}\ nat \mid mGeLUapprox \mid mELU\ real$
$\mid mELU_{taylor}\ nat\ real \mid mSELU \mid mSELU_{taylor}\ nat \mid mPReLU\ real \mid mSiLU \mid mSiLU_{taylor}\ nat$
$\mid mGaussian \mid mGaussian_{taylor}\ nat \mid mExp \mid mExp_{taylor}\ nat \mid mHardSigmoid \mid mSoftmax$
$\mid mSoftmax_{taylor}\ nat \mid mArgmax$

fun φ_{multi} :: ‹$activation_{multi} \Rightarrow (real\ list \Rightarrow real\ list)\ option$› **where**
‹$\varphi_{multi}\ mIdentity = Some\ (map\ identity)$›
| ‹$\varphi_{multi}\ mSign = Some\ (map\ sign)$›
| ‹$\varphi_{multi}\ mBinaryStep = Some\ (map\ binary\_step)$›
| ‹$\varphi_{multi}\ mSoftSign = Some\ (map\ softsign)$›
| ‹$\varphi_{multi}\ (mLogistic_{taylor}\ n\ L\ k\ v_0) = Some\ (map\ (logistic_{taylor}\ n\ L\ k\ v_0))$›
| ‹$\varphi_{multi}\ (mLogistic\ L\ k\ v_0) = Some\ (map\ (logistic\ L\ k\ v_0))$›
| ‹$\varphi_{multi}\ mSigmoid = Some\ (map\ sigmoid)$›
| ‹$\varphi_{multi}\ (mSigmoid_{taylor}\ n) = Some\ (map\ (sigmoid_{taylor}\ n))$›
| ‹$\varphi_{multi}\ mSwish = Some\ (map\ swish)$›
| ‹$\varphi_{multi}\ (mSwish_{taylor}\ n) = Some\ (map\ (swish_{taylor}\ n))$›
| ‹$\varphi_{multi}\ mTanh = Some\ (map\ tanh)$›
| ‹$\varphi_{multi}\ mReLU = Some\ (map\ relu)$›
| ‹$\varphi_{multi}\ mGeLUapprox = Some\ (map\ gelu\_approx)$›
| ‹$\varphi_{multi}\ (mGReLU\ \alpha\ m\ t) = Some\ (map\ (generalized\_relu\ \alpha\ m\ t))$›
| ‹$\varphi_{multi}\ mSoftplus = Some\ (map\ softplus)$›
| ‹$\varphi_{multi}\ (mELU\ \alpha) = Some\ (map\ (elu\ \alpha))$›
| ‹$\varphi_{multi}\ (mELU_{taylor}\ n\ \alpha) = Some\ (map\ (elu_{taylor}\ n\ \alpha))$›
| ‹$\varphi_{multi}\ mSELU = Some\ (map\ selu)$›
| ‹$\varphi_{multi}\ (mSELU_{taylor}\ n) = Some\ (map\ (selu_{taylor}\ n))$›
| ‹$\varphi_{multi}\ mExp = Some\ (map\ exp)$›
| ‹$\varphi_{multi}\ (mExp_{taylor}\ n) = Some\ (map\ (exp_{taylor}\ n))$›
| ‹$\varphi_{multi}\ mHardSigmoid = Some\ (map\ hard\_sigmoid)$›
| ‹$\varphi_{multi}\ (mPReLU\ \alpha) = Some\ (map\ (prelu\ \alpha))$›
| ‹$\varphi_{multi}\ mSiLU = Some\ (map\ silu)$›
| ‹$\varphi_{multi}\ (mSiLU_{taylor}\ n) = Some\ (map\ (silu_{taylor}\ n))$›
| ‹$\varphi_{multi}\ mGaussian = Some\ (map\ gaussian)$›
| ‹$\varphi_{multi}\ (mGaussian_{taylor}\ n) = Some\ (map\ (gaussian_{taylor}\ n))$›
| ‹$\varphi_{multi}\ mSoftmax = Some\ softmax$›
| ‹$\varphi_{multi}\ (mSoftmax_{taylor}\ n) = Some\ (softmax_{taylor}\ n)$›
| ‹$\varphi_{multi}\ mArgmax = Some\ argmax$›

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** max_{list} :: ‹'a::linorder list ⇒ 'a› **where**
‹max_{list} = Max o set›

definition min_{list} :: ‹'a::linorder list ⇒ 'a› **where**
‹min_{list} = Min o set›

lemma max_{list}_is_element: ‹$l \neq [] \Longrightarrow max_{list}\ l \in set\ l$ ›
⟨proof⟩
lemma min_{list}_is_element: ‹$l \neq [] \Longrightarrow min_{list}\ l \in set\ l$ ›
⟨proof⟩

lemma max_{list}_append_eq: ‹$max_{list}\ (xs\ @\ [x]) = max_{list}\ xs \vee max_{list}\ (xs\ @\ [x]) = x$›
⟨proof⟩
lemma min_{list}_append_eq: ‹$min_{list}\ (xs\ @\ [x]) = min_{list}\ xs \vee min_{list}\ (xs\ @\ [x]) = x$›
⟨proof⟩

lemma max_{list}_cons_eq: ‹$max_{list}\ (x\#xs) = max_{list}\ xs \vee max_{list}\ (x\#xs) = x$›
⟨proof⟩
lemma min_{list}_cons_eq: ‹$min_{list}\ (x\#xs) = min_{list}\ xs \vee min_{list}\ (x\#xs) = x$›
⟨proof⟩

lemma max_{list}_append_limit: **assumes** ‹$xs \neq []$› **shows** ‹$max_{list}\ xs \leq max_{list}\ (xs\ @\ [x])$›
⟨proof⟩
lemma min_{list}_append_limit: **assumes** ‹$xs \neq []$› **shows** ‹$min_{list}\ (xs\ @\ [x]) \leq min_{list}\ xs$›
⟨proof⟩

lemma max_{list}_cons_limit: **assumes** ‹$xs \neq []$› **shows** ‹$max_{list}\ xs \leq max_{list}\ (x\#xs)$›
⟨proof⟩
lemma min_{list}_cons_limit: **assumes** ‹$xs \neq []$› **shows** ‹$min_{list}\ (x\#xs) \leq min_{list}\ xs$›
⟨proof⟩

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

lemma *prediction2percentage_is_percentage*:
assumes ‹*0* < max_{list} *l*›
shows ‹∀ *x* ∈ *set* (*prediction2percentage l*). *x* ≤ *100.0*›
⟨proof⟩

lemma *prediction2percentage_id*: **assumes** ‹max_{list} *p* = *100*› **shows** ‹*prediction2percentage p* = *p*›
⟨*proof*⟩

lemma *prediction2percentage_min_id*:
assumes ‹*o* < max_{list} *p*›
shows ‹(*o* ≤ min_{list} (*prediction2percentage p*))= (*o* ≤ min_{list} *p*)›
⟨*proof*⟩

Maximum Prediction **definition** *posmax_of* :: ‹'*a*::*linorder list* ⇒ (*nat* × '*a*) *option*› **where**
‹*posmax_of l* = (*let m* = max_{list} *l in find* (λ *e*. *snd e* = *m*) (*enumerate o l*))›
definition *pos_of_max* :: ‹'*a*::*linorder list* ⇒ *nat option*› **where**
‹*pos_of_max l* = *map_option fst* (*posmax_of l*)›

definition *posmax_of'* :: ‹'*a*::*linorder list* ⇒ (*nat* × '*a*) *option*› **where**
‹*posmax_of' l* = (*if l* = [] *then None else Some* ((*hd o rev o* (*sort_key snd*) *o* (*enumerate o*)) *l*))›
definition *pos_of_max'* :: ‹'*a*::*linorder list* ⇒ *nat option*› **where**
‹*pos_of_max' l* = *map_option fst* (*posmax_of' l*)›

Minimum Prediction **definition** *posmin_of* :: ‹'*a*::*linorder list* ⇒ (*nat* × '*a*) *option*› **where**
‹*posmin_of l* = (*let m* = min_{list} *l in find* (λ *e*. *snd e* = *m*) (*enumerate o l*))›
definition *pos_of_min* :: ‹'*a*::*linorder list* ⇒ *nat option*› **where**
‹*pos_of_min l* = *map_option fst* (*posmin_of l*)›

definition *posmin_of'* :: ‹'*a*::*linorder list* ⇒ (*nat* × '*a*) *option*› **where**
‹*posmin_of' l* = (*if l* = [] *then None else Some* ((*hd o rev o* (*sort_key snd*) *o* (*enumerate o*)) *l*))›
definition *pos_of_min'* :: ‹'*a*::*linorder list* ⇒ *nat option*› **where**
‹*pos_of_min' l* = *map_option fst* (*posmin_of' l*)›

lemma *find_append_eq*: ‹*find P* (*xs*@[*x*]) = (*if find P xs* = *None then find P* [*x*] *else find P xs*)›
⟨*proof*⟩

lemma *posmax_of_split*: ‹*posmax_of* (*xs* @ [*x*]) = *posmax_of* (*xs*) ∨ *posmax_of* (*xs* @ [*x*]) = *Some* (*length xs*,*x*)›
⟨*proof*⟩

lemma *posmin_of_split*: ‹*posmin_of* (*xs* @ [*x*]) = *posmin_of* (*xs*) ∨ *posmin_of* (*xs* @ [*x*]) = *Some* (*length xs*,*x*)›
⟨*proof*⟩

lemma *pos_of_max_split*:
‹*pos_of_max* (*xs* @ [*x*]) = *pos_of_max* (*xs*) ∨ *pos_of_max* (*xs* @ [*x*]) = *Some* (*length xs*)›
⟨*proof*⟩

lemma *pos_of_min_split*:
‹*pos_of_min* (*xs* @ [*x*]) = *pos_of_min* (*xs*) ∨ *pos_of_min* (*xs* @ [*x*]) = *Some* (*length xs*)›
⟨*proof*⟩

lemma *posmax_of_none*: ‹(*posmax_of xs* = *None*) = (*xs* = []) ›
⟨*proof*⟩

lemma *posmin_of_none*: ‹(*posmin_of xs* = *None*) = (*xs* = []) ›
⟨*proof*⟩

lemma *posmax_of_in_snd*: ‹(*posmax_of xs*) = *Some p* ⟹ *snd p* ∈ *set xs*›

⟨*proof*⟩

lemma *posmin_of_in_snd*: ‹(*posmin_of xs*) = *Some p* ⟹ *snd p* ∈ *set xs*›
⟨*proof*⟩

lemma *pos_of_max_none*: ‹(*pos_of_max xs* = *None*) = (*xs* = []) ›
⟨*proof*⟩

lemma *pos_of_min_none*: ‹(*pos_of_min xs* = *None*) = (*xs* = []) ›
⟨*proof*⟩

lemma *take_nth_drop_eq*:
assumes ‹*xs* ≠ []›
and ‹*n* < *length xs*›
shows ‹*xs* = ((*take n xs*)@[*xs*!*n*]@(*drop* (*n*+1) *xs*))›
⟨*proof*⟩

lemma *max_in*:
assumes ‹*xs* ≠ []›
and ‹*n* < *length xs*›
and ‹∀ *x* ∈ *set* ((*take n xs*)@(*drop* (*n*+1) *xs*)). *xs*!*n* > *x*›
shows ‹*Max* (*set* ((*take n xs*)@[*xs*!*n*]@(*drop* (*n*+1) *xs*))) = ((*take n xs*)@[*xs*!*n*]@(*drop* (*n*+1) *xs*))!*n*›
⟨*proof*⟩

lemma *min_in*:
assumes ‹*xs* ≠ []›
and ‹*n* < *length xs*›
and ‹∀ *x* ∈ *set* ((*take n xs*)@(*drop* (*n*+1) *xs*)). *xs*!*n* < *x*›
shows ‹*Min* (*set* ((*take n xs*)@[*xs*!*n*]@(*drop* (*n*+1) *xs*))) = ((*take n xs*)@[*xs*!*n*]@(*drop* (*n*+1) *xs*))!*n*›
⟨*proof*⟩

lemma *max_in'*:
assumes ‹*xs* ≠ []›
and ‹*n* < *length xs*›
and ‹∀ *x* ∈ *set* ((*take n xs*)@(*drop* (*n*+1) *xs*)). *xs*!*n* > *x*›
shows ‹*Max* (*set xs*) = *xs*!*n*›
⟨*proof*⟩

lemma *min_in'*:
assumes ‹*xs* ≠ []›
and ‹*n* < *length xs*›
and ‹∀ *x* ∈ *set* ((*take n xs*)@(*drop* (*n*+1) *xs*)). *xs*!*n* < *x*›
shows ‹*Min* (*set xs*) = *xs*!*n*›
⟨*proof*⟩

lemma *snd_numerate_eq*:
xs ≠ [] ⟹ *n* < *length xs* ⟹ *j* < *n* ⟹ *snd* (*List.enumerate o xs* ! *j*) = *xs*!*j*
⟨*proof*⟩

lemma *nth_lower_max*:
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 *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** δ_{min} :: *real list* ⇒ *real* **where**
‹δ_{min} *l* = (*let m* = max_{list} *l in let m'* = max_{list} (*remove1 m l*) *in* |*m* − *m'*|)›

lemma *leq_linear_real*:
assumes *b_bound*: ‹(*b*::*real*) ∈ {*lb*..*up*}›
and *is_leq_at_bounds*: ‹((*c*$_1$ ∗ *lb* + *c*$_0$ ≤ *c*$_1$' ∗ *lb* + *c*$_0$') ∧ (*c*$_1$ ∗ *up* + *c*$_0$ ≤ *c*$_1$' ∗ *up* + *c*$_0$'))›
shows ‹*c*$_1$ ∗ *b* + *c*$_0$ ≤ *c*$_1$' ∗ *b* + *c*$_0$'›
⟨*proof*⟩

lemma *leq_linear_real'*:
assumes *b_bound*: ‹(*b*::*real*) ∈ {*lb*..*up*}›
and *is_leq_at_bounds*: ‹((*c*$_1$ ∗ *up* + *c*$_0$ ≤ *c*$_1$' ∗ *up* + *c*$_0$') ∧ (*c*$_1$ ∗ *lb* + *c*$_0$ ≤ *c*$_1$' ∗ *lb* + *c*$_0$'))›
shows ‹*c*$_1$ ∗ *b* + *c*$_0$ ≤ *c*$_1$' ∗ *b* + *c*$_0$'›
⟨*proof*⟩
lemma *le_linear_real*:
assumes *b_bound*: ‹(*b*::*real*) ∈ {*lb*..*up*}›
and *is_leq_at_bounds*: ‹((*c*$_1$ ∗ *lb* + *c*$_0$ < *c*$_1$' ∗ *lb* + *c*$_0$') ∧ (*c*$_1$ ∗ *up* + *c*$_0$ < *c*$_1$' ∗ *up* + *c*$_0$'))›

shows ‹$c_1 * b + c_0 < c_1' * b + c_0'$›
⟨*proof*⟩

lemma *le_linear_real'*:
assumes *b_bound*: ‹$(b::real) \in \{lb..up\}$›
and *is_leq_at_bounds*: ‹$((c_1 * up + c_0 < c_1' * up + c_0') \wedge (c_1 * lb + c_0 < c_1' * lb + c_0'))$›
shows ‹$c_1 * b + c_0 < c_1' * b + c_0'$›
⟨*proof*⟩

lemma *pos_max_leq'*: ‹$(pos\_of\_max\ xs = Some\ n) \Longrightarrow \forall\ x \in set\ xs.\ x \leq xs!n$›
⟨*proof*⟩

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 ‹$approx\ a\ \varepsilon\ b = (|a-b| \leq \varepsilon)$›
notation *approx* $(((\_)/ \approx[(\_)]\approx (\_))\ [60, 60]\ 60)$

fun *checkget_result_list* **where**
‹$checkget\_result\_list\ \_\ None\ None = (None, True)$›
| ‹$checkget\_result\_list\ \varepsilon\ (Some\ xs)\ (Some\ ys) = (Some\ xs, fold\ (\wedge)\ (map2\ (\lambda\ x\ y.\ x \approx[\varepsilon]\approx y)\ xs\ ys)\ True)$›
| ‹$checkget\_result\_list\ \_\ r\ \_ = (r, False)$›

definition ‹$check\_result\_list\ r\ \varepsilon\ s = snd\ (checkget\_result\_list\ \varepsilon\ r\ s)$›
notation *check_result_list* $(((\_)/ \approx[(\_)]\approx_l (\_))\ [60, 60]\ 60)$

fun *checkget_result_singleton* **where**
‹$checkget\_result\_singleton\ \_\ None\ None = (None, True)$›
| ‹$checkget\_result\_singleton\ \varepsilon\ (Some\ x)\ (Some\ y) = (Some\ x, x \approx[\varepsilon]\approx y)$›
| ‹$checkget\_result\_singleton\ \_\ r\ \_ = (r, False)$›

definition ‹$check\_result\_singleton\ r\ \varepsilon\ s = snd\ (checkget\_result\_singleton\ \varepsilon\ r\ s)$›
notation *check_result_singleton* $(((\_)/ \approx[(\_)]\approx_s (\_))\ [60, 60]\ 60)$

definition
ensure_testdata_range_list :: ‹$real \Rightarrow real\ list\ list \Rightarrow (real\ list \rightharpoonup real\ list) \Rightarrow real\ list\ list \Rightarrow bool$›
where
‹$ensure\_testdata\_range\_list\ delta\ inputs\ P\ outputs$
$= foldl\ (\wedge)\ True$
$(map\ (\lambda\ e.\ (P\ (fst\ e)) \approx[delta]\approx_l Some\ (snd\ e))$
$(zip\ inputs\ outputs))$›
notation *ensure_testdata_range_list* $((\_) \models_l \{(\_)\}\ (\_)\ \{(\_)\}\ [61, 3, 90, 3]\ 60)$

Interval Arithmetic

```
definition interval_distance :: ‹'a::{preorder,minus, zero, ord} interval ⇒ 'a interval ⇒ 'a› where
    ‹interval_distance a b = (let (la, ua) = bounds_of_interval a;
                                  (lb, ub) = bounds_of_interval b
                              in if ua ≤ lb then lb − ua
                                 else if ub ≤ la then la −ub
                                      else 0)›

fun intervals_of_list where
  ‹intervals_of_list _ [] = []›
| ‹intervals_of_list δ (x#xs) = (Interval (x− |δ|, x+|δ|))#(intervals_of_list δ xs)›

definition ‹intervals_of_l δ = map (intervals_of_list δ)›

lemma intervall_in_implies_set: (x ∈ {a..b}) ⟹ (x ∈ set_of (Interval (a,b)))
 ⟨proof⟩

lemma in_set_interval: a ≤b ⟹ (x ∈ set_of (Interval (a,b))) = (x ∈ {a..b})
 ⟨proof⟩

fun check_result_list_interval_list :: ‹'a::preorder list option ⇒ 'a interval list option ⇒ bool› where
  ‹check_result_list_interval_list None None       = True›
| ‹check_result_list_interval_list (Some xs) (Some ys) = fold (∧) (map2 (λ x y. x ∈ set_of y) xs ys) True›
| ‹check_result_list_interval_list _ _ = False›

notation check_result_list_interval_list (((_)/ ≈l (_)) [60, 60] 60)
```

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

```
definition
 ensure_testdata_interval_list :: ‹real list list ⇒ (real list ⇀ real list) ⇒ real interval list list ⇒ bool›
 where
‹ensure_testdata_interval_list inputs P outputs
  = foldl (∧) True
    (map (λ e. let a = (P (fst e)) in let b = Some (snd e) in (a ≈l b))
       (zip inputs outputs))›

notation ensure_testdata_interval_list (⊨il {(_)} (_) {(_)} [3, 90, 3] 60)
```

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

4.2.2 Maximum Classifiers

```
definition
 ensure_testdata_max_list :: ‹real list list ⇒ (real list ⇀ real list) ⇒ real list list ⇒ bool›
 where
‹ensure_testdata_max_list inputs P outputs
  = foldl (∧) True
    (map (λ e. case P (fst e) of
           None ⇒ False
         | Some p ⇒ pos_of_max p = pos_of_max (snd e))
```

(*zip inputs outputs*))›
notation *ensure_testdata_max_list* ($\models_l$ {(_)} (_) {(_)} [3, 90, 3] 60)

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

definition *ensure_delta_min* :: ‹*real* ⇒ (*real list* ⇀ *real list*) ⇒ *bool*› **where**
‹*ensure_delta_min* δ *P* = ($\forall$ *xs* ∈ *ran P*. $\delta \leq \delta_{min}$ *xs*)›
notation *ensure_delta_min* ((_) ⊨ (_) [61, 90] 60)

lemma *ensure_delta_min_dom*: ‹*ensure_delta_min* δ *P* = ($\forall$ *x* ∈ *dom P*. $\delta \leq \delta_{min}$ (*the* (*P x*)))›
⟨*proof*⟩

Further properties that we formalised can increase the confidence in the predictions of a neural network by reducing the likelihood of ambiguous classification results. This includes, e.g., the requirement that for a given input, the classification outputs have at least a given minimum distance (e.g., avoiding situations where all classification outputs show nearly identical values) shown in *ensure_delta_min*.

4.2.3 Distance-based Properties

Distance and Measurements

locale *distance* =
fixes *d*::‹'*a list* ⇒ '*a list* ⇒ ('*b*::{*linordered_ab_group_add*})›
assumes *identity*: ‹⟦*length x* = *length y*⟧ ⟹ (*d x y* = 0) = (*x* = *y*)›
and *symmetry*: ‹(*d x y* = *d y x*)›
and *triangle_inequality*: ‹⟦*length x* = *length y* ; *length z* = *length y*⟧ ⟹ (*d x z* ≤ *d x y* + *d y z*)›
begin

lemma *zero*: ‹(*d x y* = 0) = (*x* = *y*) ∨ (*length x* ≠ *length y*)›
⟨*proof*⟩

lemma ‹⟦*length x* = *length y* ; *length z* = *length y*⟧ ⟹ *d x y* + *d y x* ≥ *d x x*›
⟨*proof*⟩

lemma ‹*length x* = *length y* ⟹ 0 ≤ *d x y*›
⟨*proof*⟩

end

definition *mapfoldr* :: ‹('*a* ⇒ '*a* ⇒ '*b*) ⇒ ('*b* ⇒ '*c* ⇒ '*c*) ⇒ '*c* ⇒ '*a list* ⇒ '*a list* ⇒ '*c*› **where**
‹*mapfoldr map_f fold_f e xs ys* = *foldr fold_f* (*map2* (λ *e0 e1* . *map_f e0 e1*) *xs ys*) *e*›

definition *hamming*::‹'*a list* ⇒ '*a list* ⇒ *nat*› **where**
‹*hamming x y* = *mapfoldr* (=) (λ *e a*. *if e then a else a* + 1) 0 *x y*›

lemma *hamming_identity*: ‹*length x* = *length y* ⟹ (*hamming x y* = 0) = (*x* = *y*)›
⟨*proof*⟩

lemma *hamming_symmetry*: ‹*hamming x y* = *hamming y x*›
⟨*proof*⟩

lemma *hamming_unroll*: ⟦*length xs* = *length ys*⟧
⟹ *hamming* (*x*#*xs*) (*y*#*ys*) = (*if x* = *y then hamming xs ys else* 1 + *hamming xs ys*)
⟨*proof*⟩

lemma *hamming_triangle_inequality*:
‹⟦*length xs* = *length ys* ; *length ys* = *length zs*⟧
⟹ *hamming xs zs* ≤ *hamming xs ys* + (*hamming ys zs*)›
⟨*proof*⟩

global_interpretation *hamming_distance*: *distance hamming*
⟨*proof*⟩

definition *manhattan*::‹*real list* ⇒ *real list* ⇒ *real*› **where**
‹*manhattan* = *mapfoldr* (λ *x y* . |*x*−*y*|) (+) *o*›

lemma *manhattan_unroll*: ⟦*length xs* = *length ys*⟧
⟹ *manhattan* (*x*#*xs*) (*y*#*ys*) = |*x* − *y*| + *manhattan xs ys*
⟨*proof*⟩

lemma *manhattan_positive*: ‹*length x* = *length y* ⟹ *o* ≤ *manhattan x y*›
⟨*proof*⟩

lemma *manhattan_identity*: ‹*length x* = *length y* ⟹ (*manhattan x y* = *o*) = (*x* = *y*)›
⟨*proof*⟩

lemma *manhattan_symmetry*: ‹*manhattan x y* = *manhattan y x*›
⟨*proof*⟩

lemma *manhattan_triangle_inequality*:
‹⟦*length xs* = *length ys* ; *length ys* = *length* (*zs*::*real list*)⟧
⟹ *manhattan xs zs* ≤ *manhattan xs ys* + (*manhattan ys zs*)›
⟨*proof*⟩

global_interpretation *manhattan_distance*: *distance manhattan*
⟨*proof*⟩

definition *avg_difference*::‹*real list* ⇒ *real list* ⇒ *real*› **where**
‹*avg_difference xs ys* = (*manhattan xs ys*) / (*min* (*length xs*) (*length ys*))›

global_interpretation *avg_difference_distance*: *distance avg_difference*
⟨*proof*⟩

definition *euclidean*::‹*real list* ⇒ *real list* ⇒ *real*› **where**
‹*euclidean X Y* = *sqrt* (*mapfoldr* (λ *x y* . (*x*−*y*)²) (+) *o X Y*)›

lemma *euclidean_positive*: ‹*length x* = *length y* ⟹ *o* ≤ *euclidean x y*›
⟨*proof*⟩

lemma *euclidean_identity*: ‹*length x* = *length y* ⟹ (*euclidean x y* = *o*) = (*x* = *y*)›
⟨*proof*⟩

lemma *euclidean_symmetry*: ‹*euclidean x y* = *euclidean y x*›
⟨*proof*⟩

definition
check :: ‹('*a list* ⇒ '*a list* ⇒ '*b*) ⇒ ('*b* ⇒ *bool*) ⇒ '*a list* ⇒ ('*a list* ⇀ '*a list*)
⇒ ('*a list option* ⇒ '*a list option* ⇒ *bool*) ⇒ *bool*› **where**
‹*check d P input*$_{ref}$ *prediction P'*
= (∀ *x* ∈ *dom prediction*. *P*(*d input*$_{ref}$ *x*) ⟶ *P'* (*prediction x*) (*prediction input*$_{ref}$))›

lemma ((∀ *l* ∈ *dom prediction*. *P*(*dist i l*) ⟶ *P'* (*prediction l*) (*prediction i*)))
= ((∀ *l* ∈ {*l* ∈ *dom prediction* . *P* (*dist i l*)}. *P'* (*prediction l*) (*prediction i*)))
⟨*proof*⟩

lemma *hamming_update_1*:
length xs = *length ys* ⟹ *hamming xs ys* ≤ *1* ⟹ (∃ *i*. *xs* = *ys*[*i* := *xs*!*i*])
⟨*proof*⟩

lemma *hamming_cases1*:
assumes *l*: ‹*length xs* = *length ys*›
and *h*: ‹*hamming xs ys* ≤ *1*›
and *p*: ‹*P xs*›
and *u*: ‹⋀ *i*. *i* < *length xs* ∧ *ys* = *xs*[*i* := (*ys*!*i*)] ⟹ *P ys*›
shows ‹*P ys*›
⟨*proof*⟩

lemma *hamming_update_2*:
length xs = *length ys* ⟹ *hamming xs ys* ≤ *2* ⟹ (∃ *i j*. *xs* = (*ys*[*i* := *xs*!*i*])[*j*:= *xs*!*j*])
⟨*proof*⟩

lemma *hamming_cases2*:
assumes *l*: ‹*length xs* = *length ys*›
and *h*: ‹*hamming xs ys* ≤ *2*›
and *p*: ‹*P xs*›
and *u*: ‹⋀ *i j*. *i* < *length xs* ∧ *j* < *length xs* ∧ *ys* = *xs*[*i* := *ys*!*i*,*j* := *ys*!*j*] ⟹ *P ys*›
shows ‹*P ys*›
⟨*proof*⟩

lemma *hamming_update_n*:
length xs = *length ys* ⟹ *hamming xs ys* = *Suc n* ⟹ (∃ *i*. *hamming xs* (*ys*[*i* := *xs*!*i*]) = *n*)
⟨*proof*⟩

lemma *hamming_update_3*:
length xs = *length ys* ⟹ *hamming xs ys* ≤ *3* ⟹ (∃ *i j k*. *xs* = *ys*[*i* := *xs*!*i*,*j*:= *xs*!*j*,*k*:=*xs*!*k*])
⟨*proof*⟩

lemma *hamming_cases3*:
assumes *l*: ‹*length xs* = *length ys*›
and *h*: ‹*hamming xs ys* ≤ *3*›
and *p*: ‹*P xs*›
and *u*: ‹⋀ *i j k*. *i* < *length xs* ∧ *j* < *length xs* ∧ *k* < *length xs* ∧ *ys* = *xs*[*i* := *ys*!*i*,*j* := *ys*!*j*,*k* := *ys*!*k*] ⟹ *P ys*›

shows ‹*P ys*›
⟨*proof*⟩

end

4.3 Neural Networks as Graphs (NN_Digraph)

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

theory *NN_Digraph*
imports
Graph_Theory.Digraph
begin

definition
pipe :: ‹'*a* ⇒ ('*a* ⇒ '*b*) ⇒ '*b*› (**infixl** ‹▷› 70) **where**
‹*a* ▷ *f* = *f a*›

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

type_synonym *id* = *nat*

record ('*a*, '*b*) *Neuron* =
φ :: '*b* — activation function
α :: '*a* — learning rate
β :: '*a* — bias
uid :: *id* — unique identifier

datatype ('*a*, '*b*) *neuron* = *In id* | *Out id* | *Neuron* ‹('*a*, '*b*) *Neuron*›

fun *uid* **where**
‹*uid* (*In nid*) = *nid*›
| ‹*uid* (*Out nid*) = *nid*›
| ‹*uid* (*Neuron n*) = *Neuron.uid n*›

record ('*a*, '*b*) *edge* =
ω :: '*a* — weight input to head
tl :: ‹('*a*, '*b*) *neuron*› — source neuron
hd :: ‹('*a*, '*b*) *neuron*› — target neuron

type_synonym ('*a*, '*b*) *nn_pregraph* = ‹(('*a*, '*b*) *neuron*, ('*a*, '*b*) *edge*) *pre_digraph*›

definition *upd_edge* :: ‹('*a*, '*b*) *nn_pregraph* ⇒ (('*a*, '*b*) *edge* ⇒ ('*a*, '*b*) *edge*)
⇒ ('*a*, '*b*) *nn_pregraph*› **where**
‹*upd_edge G upd* = (|
verts = *verts G* ,
arcs = *upd* ' (*arcs G*),
tail = *tail G*,
head = *head G*
|)›

definition ‹*upd*$_\omega$ ω' *hd_n*$_{id}$ *tl_n*$_{id}$ *a* = (*if uid* (*hd a*) = *hd_n*$_{id}$ ∧ *uid* (*tl a*) = *tl_n*$_{id}$
then (|ω = ω', *tl* = *tl a*, *hd* = *hd a* |)
else a)›

definition *upd_neuron* :: ‹(*'a*, *'b*) *nn_pregraph* ⇒ ((*'a*, *'b*) *Neuron* ⇒ (*'a*, *'b*) *Neuron*)
⇒ (*'a*, *'b*) *nn_pregraph* › **where**
‹*upd_neuron G upd* = (*let upd_Neuron* = *case_neuron In Out* (λ *n*. *Neuron* (*upd n*))
in (|
verts = *upd_Neuron* ‘ (*verts G*) ,
arcs = (λ *a*. (| ω = ω *a*,
tl = *upd_Neuron* (*tl a*),
hd = *upd_Neuron* (*hd a*)|)) ‘ (*arcs G*),
tail = *tail G*,
head = *head G*
|))›

definition ‹*upd*$_\varphi$ φ' n_{id} *n* = (*if Neuron.uid n* = n_{id}
then (|$\varphi = \varphi'$, $\alpha = \alpha$ *n*, $\beta = \beta$ *n*, *uid* = *Neuron.uid n* |)
else n)›

definition ‹*upd*$_\beta$ β' n_{id} *n* = (*if Neuron.uid n* = n_{id}
then (|$\varphi = \varphi$ *n*, $\alpha = \alpha$ *n*, $\beta = \beta'$, *uid* = *Neuron.uid n* |)
else n)›

definition ‹*upd*$_\alpha$ α' n_{id} *n* = (*if Neuron.uid n* = n_{id}
then (|$\varphi = \varphi$ *n*, $\alpha = \alpha'$, $\beta = \beta$ *n*, *uid* = *Neuron.uid n* |)
else n)›

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

definition *input_verts* :: ‹((*'a*, *'b*) *neuron*, (*'a*, *'b*) *edge*) *pre_digraph* ⇒ (*'a*, *'b*) *neuron set*›
where
‹*input_verts G* = (*verts G*) − (*hd* ‘ *arcs G*)›

definition *output_verts* :: ‹((*'a*, *'b*) *neuron*, (*'a*, *'b*) *edge*) *pre_digraph* ⇒ (*'a*, *'b*) *neuron set*›
where
‹*output_verts G* = (*verts G*) − (*tl* ‘ *arcs G*)›

definition *internal_verts* :: ‹((*'a*, *'b*) *neuron*, (*'a*, *'b*) *edge*) *pre_digraph* ⇒ (*'a*, *'b*) *neuron set*›
where
‹*internal_verts G* = (*verts G*) − ((*input_verts G*) ∪ (*output_verts G*))›

locale *nn_pregraph* = *digraph G*
for *G*::‹((*'a*::{*comm_monoid_add*,*times*,*linorder*}, *'b*) *neuron*, (*'a*, *'b*) *edge*) *pre_digraph*› +
assumes *id_vert_inj*: ‹*inj_on uid* (*verts G*)›
and *tail_eq_tl*: ‹*tail G* = *tl*›
and *head_eq_hd*: ‹*head G* = *hd*›
and *ids_growing*: ‹∀ *e* ∈ *arcs G*. *uid* (*tl e*) < *uid* (*hd e*)› — Not strictly necessary, but simplifies termination proofs.
begin

lemma *nn_pregraph*: *nn_pregraph G* ⟨*proof*⟩

end

definition ‹*uids G* = *uid* ‘ *verts G*›

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*: ‹*nn_pregraph* (*add_nn_edge G a*)›
⟨*proof*⟩

The operation *del_arc* preserves neural networks

lemma *nn_pregraph_del_arc*: ‹*nn_pregraph* (*del_arc a*)›
⟨*proof*⟩

end

4.3.3 Updating Neurons

context *nn_pregraph* **begin**

lemma *upd*$_\varphi$*_nid_immutable*[*simp*]: ‹*Neuron.uid n* $\neq$ n_{id} $\Longrightarrow$ *n* = (*upd*$_\varphi$ φ' n_{id} *n*)›
and *upd*$_\varphi$*_id_immutable*[*simp*]: ‹*Neuron.uid n* = *Neuron.uid* (*upd*$_\varphi$ φ' n_{id} *n*)›
and *upd*$_\varphi$*_*α*_immutable*[*simp*]: ‹α *n* = α (*upd*$_\varphi$ φ' n_{id} *n*)›
and *upd*$_\varphi$*_*β*_immutable*[*simp*]: ‹β *n* = β (*upd*$_\varphi$ φ' n_{id} *n*)›
and *upd*$_\beta$*_nid_immutable*[*simp*]: ‹*Neuron.uid n* $\neq$ n_{id} $\Longrightarrow$ *n* = (*upd*$_\beta$ β' n_{id} *n*)›
and *upd*$_\beta$*_id_immutable*[*simp*]: ‹*Neuron.uid n* = *Neuron.uid* (*upd*$_\beta$ β' n_{id} *n*)›
and *upd*$_\beta$*_*φ*_immutable*[*simp*]: ‹φ *n* = φ (*upd*$_\beta$ β' n_{id} *n*)›
and *upd*$_\beta$*_*α*_immutable*[*simp*]: ‹α *n* = α (*upd*$_\beta$ β' n_{id} *n*)›
and *upd*$_\alpha$*_nid_immutable*[*simp*]: ‹*Neuron.uid n* $\neq$ n_{id} $\Longrightarrow$ *n* = (*upd*$_\alpha$ α' n_{id} *n*)›
and *upd*$_\alpha$*_id_immutable*[*simp*]: ‹*Neuron.uid n* = *Neuron.uid* (*upd*$_\alpha$ α' n_{id} *n*)›
and *upd*$_\alpha$*_*φ*_immutable*[*simp*]: ‹φ *n* = φ (*upd*$_\alpha$ α' n_{id} *n*)›
and *upd*$_\alpha$*_*β*_immutable*[*simp*]: ‹β *n* = β (*upd*$_\alpha$ α' n_{id} *n*)›
⟨*proof*⟩

lemma *wf_digraph_update_neuron*:
assumes ‹$\forall$ *n*. *Neuron.uid n* = *Neuron.uid* (*upd n*)›
shows‹*wf_digraph* (*upd_neuron G upd*)›
⟨*proof*⟩

lemma *fin_digraph_update_neuron*:
assumes ‹$\forall$ *n*. *Neuron.uid n* = *Neuron.uid* (*upd n*)›
shows‹*fin_digraph* (*upd_neuron G upd*)›
⟨*proof*⟩

lemma *nomulti_digraph_update_neuron*:
assumes ‹$\forall$ *n*. *Neuron.uid n* = *Neuron.uid* (*upd n*)›
shows ‹*nomulti_digraph* (*upd_neuron G upd*)›
⟨*proof*⟩

lemma *loopfree_digraph_update_neuron*:
assumes ‹$\forall$ *n*. *Neuron.uid n* = *Neuron.uid* (*upd n*)›
shows ‹*loopfree_digraph* (*upd_neuron G upd*)›
⟨*proof*⟩

lemma *nn_pregraph_update_neuron*:

assumes ‹∀ *n*. *Neuron.uid n* = *Neuron.uid* (*upd n*)›
shows‹*nn_pregraph* (*upd_neuron G upd*)›
⟨*proof*⟩

lemma *nn_pregraph_upd*$_\varphi$[*simp*]: ‹*nn_pregraph* (*upd_neuron G* (*upd*$_\varphi$ φ' n_{id}))›
and *nn_pregraph_upd*$_\beta$[*simp*]: ‹*nn_pregraph* (*upd_neuron G* (*upd*$_\beta$ β' n_{id}))›
and *nn_pregraph_upd*$_\alpha$[*simp*]: ‹*nn_pregraph* (*upd_neuron G* (*upd*$_\alpha$ α' n_{id}))›
⟨*proof*⟩

end

4.3.4 Updating arcs (edges)

context *nn_pregraph* **begin**

lemma *upd*$_\omega$*_tl_immutable*[*simp*]: ‹(*tl a* = *tl* (*upd*$_\omega$ ω' n_{hd} n_{tl} *a*))›
and *upd*$_\omega$*_hd_immutable*[*simp*]: ‹(*hd a* = *hd* (*upd*$_\omega$ ω' n_{hd} n_{tl} *a*))›
⟨*proof*⟩

lemma *upd*$_\omega$*_ends_immutable*[*simp*]: ‹(*arc_to_ends G a* = *arc_to_ends G* (*upd*$_\omega$ ω' n_{hd} n_{tl} *a*))›
⟨*proof*⟩

lemma *upd_edge_tail_immutable*:
‹*tail* (*upd_edge G upd*) = *tail G* ›
⟨*proof*⟩

lemma *upd_edge_head_immutable*:
‹*head* (*upd_edge G upd*) = *head G* ›
⟨*proof*⟩

lemma *upd_edge_vert_immutable*: ‹*verts* (*upd_edge G upd*) = *verts G*›
⟨*proof*⟩

lemma *upd_edge_arcs*: ‹*a* ∈ *arcs* (*upd_edge G upd*) ⟹ ∃ *x* ∈ *arcs G*. *a* = *upd x*›
⟨*proof*⟩

lemma *wf_digraph_update_edge*:
assumes ‹∀ *a* ∈ *arcs G*. (*arc_to_ends G a* = *arc_to_ends G* (*upd a*))›
shows ‹*wf_digraph* (*upd_edge G upd*)›
⟨*proof*⟩

lemma *fin_digraph_update_edge*:
assumes ‹∀ *a* ∈ *arcs G*. (*arc_to_ends G a* = *arc_to_ends G* (*upd a*))›
shows ‹*fin_digraph* (*upd_edge G upd*)›
⟨*proof*⟩

lemma *nomulti_digraph_update_edge*:
assumes ‹∀ *a* ∈ *arcs G*. (*arc_to_ends G a* = *arc_to_ends G* (*upd a*))›
shows ‹*nomulti_digraph* (*upd_edge G upd*)›
⟨*proof*⟩

lemma *loopfree_digraph_update_edge*:

assumes ‹∀ *a* ∈ *arcs G*. (*arc_to_ends G a* = *arc_to_ends G* (*upd a*))›
shows ‹*loopfree_digraph* (*upd_edge G upd*)›
⟨*proof*⟩

lemma *nn_pregraph_update_edge*:
assumes ‹∀ *a* ∈ *arcs G*. (*arc_to_ends G a* = *arc_to_ends G* (*upd a*))›
and ‹∀ *a* ∈ *arcs G*. *uid* (*tl* (*upd a*)) < *uid* (*hd* (*upd a*))›
shows ‹*nn_pregraph* (*upd_edge G upd*)›
⟨*proof*⟩

lemma *nn_pregraph_upd*$_\omega$[*simp*]: ‹*nn_pregraph* (*upd_edge G* (*upd*$_\omega$ *ω*′ n_{hd} n_{tl}))›
⟨*proof*⟩

end

record ('*a*, '*b*, '*c*) *neural_network* =
graph :: ((′*a*, ′*b*) *neuron*, (′*a*, ′*b*) *edge*) *pre_digraph*
activation_tab :: ‹′*b* ⇒ ′*c option*›

definition *upd_edge*′ :: ‹(′*a*, ′*b*, ′*c*) *neural_network* ⇒ ((′*a*, ′*b*) *edge* ⇒ (′*a*, ′*b*) *edge*)
⇒ (′*a*, ′*b*, ′*c*) *neural_network*› **where**
‹*upd_edge*′ *N upd* = (|
graph = *upd_edge* (*graph N*) *upd*,
activation_tab = *activation_tab N*
|)›

definition *upd_neuron*′ :: ‹(′*a*, ′*b*, ′*c*) *neural_network* ⇒ ((′*a*, ′*b*) *Neuron* ⇒ (′*a*, ′*b*) *Neuron*)
⇒ (′*a*, ′*b*, ′*c*) *neural_network*› **where**
‹*upd_neuron*′ *N upd* = (|
graph = *upd_neuron* (*graph N*) *upd*,
activation_tab = *activation_tab N*
|)›

definition *input_layer* :: ‹(′*a*, ′*b*, ′*c*) *neural_network* ⇒ (′*a*, ′*b*) *neuron set*› **where**
‹*input_layer N* = *input_verts* (*graph N*)›

definition *arity* :: ‹(′*a*, ′*b*, ′*c*) *neural_network* ⇒ *nat*› **where**
‹*arity N* = *card* (*input_layer N*)›

definition *input_layer_ids* :: ‹(′*a*, ′*b*, ′*c*) *neural_network* ⇒ *id set*› **where**
‹*input_layer_ids N* = *uid* ‘ (*input_layer N*)›

definition *output_layer* :: ‹(′*a*, ′*b*, ′*c*) *neural_network* ⇒ (′*a*, ′*b*) *neuron set*› **where**
‹*output_layer N* = *output_verts* (*graph N*)›

definition *output_layer_ids* :: ‹(′*a*, ′*b*, ′*c*) *neural_network* ⇒ *id set*› **where**
‹*output_layer_ids N* = *uid* ‘ (*output_layer N*)›

definition *incoming_arcs* :: ‹(′*a*, ′*b*, ′*c*) *neural_network* ⇒ *id* ⇒ (′*a*, ′*b*) *edge set*› **where**

‹*incoming_arcs N* n_{id} = {*a* . *a* ∈ *arcs* (*graph N*) ∧ *uid* (*hd a*) = n_{id}}›

definition ‹*sorted_list_of_set'* ≡ *map_fun id id* (*folding_on.F* (*insort_key* (λ *x*. *uid* (*tl x*)))) []›

definition *incoming_arcs_l* :: ‹('*a*, '*b*, '*c*) *neural_network* ⇒ *id* ⇒ ('*a*, '*b*) *edge list*› **where**
‹*incoming_arcs_l N* n_{id} = *sorted_list_of_set'* (*incoming_arcs N* n_{id})›

context *nn_pregraph* **begin**
lemma *incoming_arcs_l_eq_incoming_arcs*: ‹*set* (*incoming_arcs_l N* n_{id})= (*incoming_arcs N* n_{id})›
⟨*proof*⟩

lemma *incoming_arcs_l_alt_def*: ‹ (*incoming_arcs_l N* n_{id})
= (*sorted_key_list_of_set* (λ *x*. *uid* (*tl x*)) (*incoming_arcs N* n_{id}))›
⟨*proof*⟩

lemma *insert_key_comm*: *inj f* ⟹ (*insort_key f y* ∘ *insort_key f x*) = (*insort_key f x* ∘ *insort_key f y*)
⟨*proof*⟩

lemma *tl_subset_verts*: ‹*tl* ' (*arcs G*) ⊆ *verts G* ›
⟨*proof*⟩

lemma *hd_subset_verts*: ‹*hd* ' (*arcs G*) ⊆ *verts G* ›
⟨*proof*⟩

lemma *inj_on_tl*: ‹*inj_on uid* (*tl* ' (*arcs G*)) ›
⟨*proof*⟩

end

definition *outgoing_arcs* :: ‹('*a*, '*b*, '*c*) *neural_network* ⇒ *id* ⇒ ('*a*, '*b*) *edge set*› **where**
‹*outgoing_arcs N* n_{id} = {*a* . *a* ∈ *arcs* (*graph N*) ∧ *uid* (*tl a*) = n_{id}}›

definition *neurons* :: ‹('*a*, '*b*, '*c*) *neural_network* ⇒ ('*a*, '*b*) *neuron set*› **where**
‹*neurons* = *verts o graph*›

definition *edges* :: ‹('*a*, '*b*, '*c*) *neural_network* ⇒ ('*a*, '*b*) *edge set*› **where**
‹*edges* = *arcs o graph*›

locale *nn_graph* = *nn_pregraph* +
assumes *id_vert_inj*: ‹*inj_on uid* (*verts G*)›
and *inputs_In*:
‹*input_verts G* = *Set.filter* (λ *v*. (*case v of In _* ⇒ *True* | *_* ⇒ *False*)) (*verts G*) ›
and *outputs_Out*:
‹*output_verts G* = *Set.filter* (λ *v*. (*case v of Out _* ⇒ *True* | *_* ⇒ *False*)) (*verts G*) ›
and *internal_Neuron*:
‹*internal_verts G* = *Set.filter* (λ *v*. (*case v of Neuron _* ⇒ *True* | *_* ⇒ *False*)) (*verts G*) ›
begin

lemma *nn_graph*: *nn_graph G* ⟨*proof*⟩
end

locale *neural_network_digraph* =

fixes *N*::‹('*a*::{*comm_monoid_add*,*times*,*linorder*,*one*}, '*b*, '*c*) *neural_network*›
assumes ‹*nn_graph* (*graph N*)›
and ‹φ ' {*n* . *Neuron n* ∈ (*verts* (*graph N*)) } ⊆ *dom* (*activation_tab N*)›

4.3.5 The empty neural network

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

lemma *nn_pregraph_empty*[*simp*]: ‹*nn_pregraph* (*empty*)›
⟨*proof*⟩

lemma *nn_graph_empty*[*simp*]: ‹*nn_graph* (*empty*)›
⟨*proof*⟩

lemma *fold_inv*: *P e* ⟹ (∀ *e' x*. *P e'* ⟶ *P* (*f x e'*)) ⟹ *P* (*fold f xs e*)
⟨*proof*⟩

lemma *nn_pregraph_fold*: ‹*nn_pregraph G* ⟹ *nn_pregraph* (*foldr* (λ *a b*. *add_nn_edge b a*) *edge_list G*)›
⟨*proof*⟩

definition
‹*mk_nn_pregraph edge_list* = *foldr* (λ *a b*. *add_nn_edge b a*) *edge_list empty*›

lemma *nn_pregraph_mk*: ‹*nn_pregraph*(*mk_nn_pregraph edge_list*)›
⟨*proof*⟩

lemma *verts_subseteq_add_edge*: *nn_pregraph G* ⟹ *verts G* ⊆ *verts* (*add_nn_edge G a*)
⟨*proof*⟩

4.3.6 Computing Predictions of Neural Networks

datatype *error* = *OK* | *ERROR*

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

function (*sequential*, *domintros*) *predict*$_{digraph\_single}$'::‹*nat*
⇒ ('*a*::{*comm_monoid_add*,*times*,*linorder*,*one*}, '*b*, '*a* ⇒ '*a*) *neural_network*
⇒ (*id* ⇀ '*a*) ⇒ ('*a*, '*b*) *edge* ⇒ ('*a* × *error*)›
where
‹*predict*$_{digraph\_single}$' _ *N inputs* (⦇ω=_, *tl*= _, *hd*=*In* _⦈) = (*o*, *ERROR*)›
| ‹*predict*$_{digraph\_single}$' _ *N inputs* (⦇ω=_, *tl*=*Out* _ , *hd*=_ ⦈) = (*o*, *ERROR*)›
| ‹*predict*$_{digraph\_single}$' _ *N inputs* (⦇ω=ω', *tl*=*In uid*$_{in}$, *hd*=_ ⦈) = (*case inputs uid*$_{in}$ *of*
None ⇒ (*o*, *ERROR*)
| *Some v* ⇒ (*v* ∗ ω', *OK*))›
| ‹*predict*$_{digraph\_single}$' *n N inputs e* = (*if o* < *n then*
(*let*
ω' = ω *e*;
tl' = (*case* (*tl e*) *of* (*Neuron t*) ⇒ *t*);
E' = *incoming_arcs N* (*Neuron.uid tl'*);
lvals = ((λ *e'*. (*case predict*$_{digraph\_single}$' (*n*−1) *N inputs e' of*

$(\_, ERROR) \Rightarrow ((o,o),\ ERROR)$
$\mid (v, OK) \ \Rightarrow ((v\ ,uid\ (tl\ e')), OK)))\ `\ E')$
in
$($ *case* $(activation\_tab\ N)\ (\varphi\ tl')$ *of*
Some $f \Rightarrow (\omega' * (\ f((\sum\ v \in lvals.\ (fst\ (fst\ v))) + (\beta\ tl'))),\ OK)$
$\mid$ *None* $\Rightarrow (o, ERROR)))$
else $(o, ERROR)\ )$›
⟨*proof*⟩

termination
⟨*proof*⟩

definition
‹$predict_{digraph\_single}$ *N inputs e* = (*case* $predict_{digraph\_single}{}'$ (*card* (*edges N*)) *N inputs e of*
$(r, OK) \Rightarrow$ *Some r*
$\mid (\_, ERROR) \Rightarrow$ *None*)›

definition
‹*get_input_neuron_ids_l N* = *sorted_list_of_set* (*uid* ` (*input_verts* (*graph N*)))›
definition
‹*mk_input_map N vs* = *map_of* (*rev* (*zip* (*get_input_neuron_ids_l N*) *vs*))›
definition
‹*get_output_edge_ids_l N* = *sorted_list_of_set* (*uid* ` (*output_verts* (*graph N*)))›
definition
‹*get_output_edge_l N* = *map the_elem* (*map* (λ *i*. {*e* . *e* ∈ *edges N* ∧ *i* = *uid* (*hd e*)}) (*get_output_edge_ids_l N*))›
definition
‹ $predict_{digraph\_single\_list}$ *N inputs'* = *those* (*map* (λ *e*. $predict_{digraph\_single}$ *N* (*mk_input_map N inputs'*) *e*) (*get_output_edge_l N*))›

context *neural_network_digraph_single* **begin**

lemma *ids_growing'*: ‹*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
‹$predict_{digraph}$ _ _ = ([], *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} :: ‹'a::linorder Matrix.mat ⇒ 'a› **where**
‹max_{mat} = Max o elements_mat›

definition min_{mat} :: ‹'a::linorder Matrix.mat ⇒ 'a› **where**
‹min_{mat} = Min o elements_mat›

lemma *finite_elements_mat*: *finite* (*elements_mat A*)
⟨*proof*⟩

lemma max_{mat}*_is_element*:
shows ‹*elements_mat m* ≠ {} ⟹ max_{mat} *m* ∈ *elements_mat m*›
⟨*proof*⟩

lemma min_{mat}*_is_element*:
‹*elements_mat m* ≠ {} ⟹ min_{mat} *m* ∈ *elements_mat m*›
⟨*proof*⟩

definition *max_list* :: 'a::linorder list ⇒ 'a **where**
max_list xs = *fold max xs* (*hd xs*)

definition *min_list* :: 'a::linorder list ⇒ 'a **where**
min_list xs = *fold min xs* (*hd xs*)

definition max_{vec} :: ‹'a::linorder Matrix.vec ⇒ 'a› **where**
‹max_{vec} = max_list o list_of_vec›

definition min_{vec} :: ‹'a::linorder Matrix.vec ⇒ 'a› **where**
‹min_{vec} = min_list o list_of_vec›

lemma max_{vec}*_is_element*:
shows ‹*list_of_vec m* ≠ [] ⟹ max_{vec} *m* ∈ *set*(*list_of_vec m*)›
⟨*proof*⟩

lemma min_{vec}_is_element:
shows ‹list_of_vec m ≠ [] ⟹ min_{vec} m ∈ set(list_of_vec m)›
⟨proof⟩

lemma max_{vec}_vCons_append_eq: ‹max_{vec} (vCons x xs) = max_{vec} xs ∨ max_{vec} (vCons x xs) = x›
⟨proof⟩

lemma max_{vec}_append_eq: ‹max_{vec} (vec_of_list (xs @ [x])) = max_{vec} (vec_of_list xs) ∨ max_{vec} (vec_of_list (xs @ [x])) = x›
⟨proof⟩

lemma min_{vec}_vCons_append_eq: ‹min_{vec} (vCons x xs) = min_{vec} xs ∨ min_{vec} (vCons x xs) = x›
⟨proof⟩

lemma min_{vec}_append_eq: ‹min_{vec} (vec_of_list (xs @ [x])) = min_{vec} (vec_of_list xs) ∨ min_{vec} (vec_of_list (xs @ [x])) = x›
⟨proof⟩

lemma max_{vec}_vec_cons_eq: ‹max_{vec} ((vec_of_list [x]) @$_v$ xs) = max_{vec} xs ∨ max_{vec} ((vec_of_list [x]) @$_v$ xs) = x›
⟨proof⟩

lemma max_{vec}_cons_eq: ‹max_{vec} (vec_of_list (x#xs)) = max_{vec} (vec_of_list xs) ∨ max_{vec} (vec_of_list (x#xs)) = x›
⟨proof⟩

lemma min_{vec}_vec_cons_eq: ‹min_{vec} ((vec_of_list [x]) @$_v$ xs) = min_{vec} xs ∨ min_{vec} ((vec_of_list [x]) @$_v$ xs) = x›
⟨proof⟩

lemma min_{list}_cons_eq: ‹min_{vec} (vec_of_list (x#xs)) = min_{vec} (vec_of_list xs) ∨ min_{vec} (vec_of_list (x#xs)) = x›
⟨proof⟩

lemma max_{vec}_vec_append_limit: **assumes** ‹xs ≠ vNil› **shows** ‹max_{vec} xs ≤ max_{vec} (vCons x xs)›
⟨proof⟩

lemma max_{vec}_append_limit: **assumes** ‹xs ≠ []› **shows** ‹max_{vec} (vec_of_list xs) ≤ max_{vec} (vec_of_list (xs @ [x]))›
⟨proof⟩

lemma min_{vec}_vec_append_limit: **assumes** ‹xs ≠ vNil› **shows** ‹min_{vec} xs ≥ min_{vec} (vCons x xs)›
⟨proof⟩

lemma min_{vec}_append_limit: **assumes** ‹xs ≠ []› **shows** ‹min_{vec} (vec_of_list xs) ≥ min_{vec} (vec_of_list (xs @ [x]))›
⟨proof⟩

lemma max_{vec}_vec_cons_limit: **assumes** ‹xs ≠ vNil› **shows** ‹max_{vec} xs ≤ max_{vec} ((vec_of_list [x]) @$_v$ xs)›
⟨proof⟩

lemma max_{vec}_cons_limit: **assumes** ‹xs ≠ []› **shows** ‹max_{vec} (vec_of_list xs) ≤ max_{vec} (vec_of_list (x#xs))›
⟨proof⟩

lemma min_{vec}_vec_cons_limit: **assumes** ‹xs ≠ vNil› **shows** ‹min_{vec} xs ≥ min_{vec} ((vec_of_list [x]) @$_v$ xs)›
⟨proof⟩

lemma min_{vec}_cons_limit: **assumes** ‹xs ≠ []› **shows** ‹min_{vec} (vec_of_list xs) ≥ min_{vec} (vec_of_list (x#xs))›
⟨proof⟩

Converting Predictions to Percentages definition *prediction2percentage* :: ‹*real Matrix.vec* ⇒ *real Matrix.vec*› where
‹*prediction2percentage m* = (let *m'* = max_{vec} *m* in *map_vec* (λ *e*. *e* / *m'* ∗ *100.0*) *m*)›

lemma *prediction2percentage_id*:
assumes ‹max_{vec} *p* = *100*›
shows ‹*prediction2percentage p* = *p*›
⟨*proof*⟩

Maximum Prediction definition *posmax_of* :: ‹'*a*::*linorder Matrix.vec* ⇒ (*nat* × '*a*) *option*› where
‹*posmax_of l* = (let *m* = max_{vec} *l* in *find* (λ *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*) (λ *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* (λ (*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* (λ *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* ⇒ *real Matrix.vec list* ⇒ (*real Matrix.vec* ⇀ *real Matrix.vec*) ⇒ *real Matrix.vec list* ⇒ *bool*›
where
‹*ensure_testdata_range_mat delta inputs P outputs*
= *foldl* (∧) *True*
(*map* (λ *e*. (*P* (*fst e*)) ≈[*delta*]≈$_m$ *Some* (*snd e*))
(*zip inputs outputs*))›
notation *ensure_testdata_range_mat* ((_) ⊨$_m$ {(_)} (_) {(_)} [61, 3, 90, 3] 60)

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

definition ‹*intervals_of_m* δ = *map* (*intervals_of_mat* δ) ›

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

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

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

definition *ensure_testdata_interval_mat* :: ‹*real Matrix.vec list* ⇒ (*real Matrix.vec* ⇀ *real Matrix.vec*) ⇒ *real interval Matrix.vec list* ⇒ *bool*›
where
‹*ensure_testdata_interval_mat inputs P outputs*
= *foldl* (∧) *True*
(*map* (λ *e* . *let a* = (*P* (*fst e*)) *in let b* = *Some* (*snd e*) *in* (*a* ≈$_m$ *b*))
(*zip inputs outputs*)) ›
notation *ensure_testdata_interval_mat* (⊨$_{im}$ {(_)} (_) {(_)} [3, 90, 3] 60)

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

Maximum Classifiers

definition
ensure_testdata_max_mat :: ‹*real Matrix.vec list* ⇒ (*real Matrix.vec* ⇀ *real Matrix.vec*) ⇒ *real Matrix.vec list* ⇒ *bool*›
where
‹*ensure_testdata_max_mat inputs P outputs*
= *foldl* (∧) *True*
(*map* (λ *e*. *case P* (*fst e*) *of*
None ⇒ *False*
| *Some p* ⇒ *pos_of_max p* = *pos_of_max* (*snd e*))
(*zip inputs outputs*))›
notation *ensure_testdata_max_mat* (⊨$_m$ {(_)} (_) {(_)} [3, 90, 3] 60)

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

end

5.1.3 Sequential Layers (NN_Layers)

theory
NN_Layers
imports
Activation_Functions
begin

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

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

record $('b)$ *ActivationRecord* = *InOutRecord* +
φ :: $'b$

record $('a, 'b, 'c)$ *LayerRecord* = ‹$('b)$ *ActivationRecord*› +
β :: ‹$'a$›
ω :: ‹$'c$›

datatype $('a, 'b, 'c)$ *layer* = *In* ‹*InOutRecord*›
| *Out* ‹*InOutRecord*›
| *Dense* ‹$('a, 'b, 'c)$ *LayerRecord*›
| *Activation* ‹$('b)$ *ActivationRecord*›

fun $units_l$ **where**
‹$units_l$ $(In\ l) = units\ l$›
| ‹$units_l$ $(Out\ l) = units\ l$›
| ‹$units_l$ $(Dense\ l) = units\ l$›
| ‹$units_l$ $(Activation\ l) = units\ l$›

lemmas $[nn\_layer] = InOutRecord.simps\ ActivationRecord.simps\ LayerRecord.simps\ layer.simps\ units_l.simps$

The datatype *layer* models the currently supported layer types

As we are using a representation of a network as a list of layers, we also support different layer types and their computations. Currently, our sequential layers model supports five layer types *In input* (input layer), *Out output* (output layer), *Dense dense_layer* (dense layer), and *Activation activation_layer* (activation layer). As we allow for the abstraction of activation functions, we abstract from the actual type for the activation function (modelled by the type variable $'b$ and from the actual type of weight and bias (modelled by the type variables $'a$ and $'c$ respectively).

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

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

fun *isIn* **where**
‹*isIn* $(In\ \_)$ = *True*›
| ‹*isIn* _ = *False*›

fun *isOut* **where**

```
  ‹isOut (Out _) = True›
| ‹isOut _      = False›

fun isInternal where
  ‹isInternal (Out _) = False›
| ‹isInternal (In _)  = False›
| ‹isInternal _      = True›

lemma isInternal': ‹isInternal n = (¬ (isIn n) ∧ ¬ (isOut n))›
⟨proof⟩

record ('a, 'b,'c) neural_network_seq_layers =
 layers :: ‹('a, 'b, 'c) layer  list›
 activation_tab :: ‹'b ⇒ (('a ⇒ 'a ) option)›

lemmas [nn_layer] = neural_network_seq_layers.simps
```

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

```
fun  out_deg_layer
where
  ‹out_deg_layer (In l) = (units l)›
| ‹out_deg_layer (Out l) = (units l)›
| ‹out_deg_layer (Activation l) = units l›
| ‹out_deg_layer (Dense l) = units l›

fun units_layer where
  ‹units_layer (In l) = units l›
| ‹units_layer (Out l) = units l›
| ‹units_layer (Activation l) = units l›
| ‹units_layer (Dense l) = units l›

fun φ_layer where
  ‹φ_layer (In l) = None›
| ‹φ_layer (Out l) = None›
| ‹φ_layer (Activation l) = Some (φ l)›
| ‹φ_layer (Dense l) = Some (φ 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 (ω 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 ∘ last ∘ 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−lipschitz_on* (*X*::*real set*) (*relu*)
⟨*proof*⟩

lemma *identity_lipschitz*: *1−lipschitz_on* (*X*::*real set*) (*identity*)
⟨*proof*⟩

Neural Network: Layers **lemma** *input_output_lipschitz_continuous*:
‹*1−lipschitz_on* (*U*::*real set*) (λ *i*. *i*)›
⟨*proof*⟩

lemma *activation_lipschitz_continuous*:
assumes ‹*C−lipschitz_on U f*›
shows ‹*C−lipschitz_on U* (λ *i* . *f i*)›
⟨*proof*⟩

lemma *lipschitz_on_add_const*:
shows (*1*::*real*)*−lipschitz_on* (*U*::*real set*) (λ*x*. *x* + *c*)
⟨*proof*⟩

lemma *lipschitz_on_fold_add*:
shows *1−lipschitz_on* (*U*::*real set*) (*fold* (+) *xs*)
⟨*proof*⟩

lemma *lipschitz_on_fold_add_zero*:
shows *1−lipschitz_on* (*U*::*real set*) (λ *x* . *fold* (+) [*x*] (*0*::*real*))
⟨*proof*⟩

lemma *lipschitz_on_foldr_add*:
shows *1−lipschitz_on* (*U*::*real set*) (λ *s*. *foldr* (+) *xs s*)
⟨*proof*⟩

lemma *lipschitz_on_sumlist_rev*:
shows *1*−*lipschitz_on* (*U*::*real set*) ((+) (*sum_list* (*rev xs*)))
⟨*proof*⟩

lemma *lipschitz_on_sumlist*:
shows *1*−*lipschitz_on* (*U*::*real set*) ((+) (*sum_list xs*))
⟨*proof*⟩

lemma *lipschitz_on_mult_const*:
shows |*c*|−*lipschitz_on* (*U*::*real set*) (λ *x* . *x* ∗ *c*)
⟨*proof*⟩

lemma *lipschitz_on_weighted_sum_single*:
|*w*|−*lipschitz_on* (*U*::*real set*) (λ *x* . *x* ∗ *w* + *b*)
⟨*proof*⟩

lemma *lipschitz_on_fold_add_zero'*:
shows *2*−*lipschitz_on* (*U*::*real set*) (λ *x* . (*fold* (+) [*x*,*x*] (*o*::*real*)) + *w*)
⟨*proof*⟩

lemma *lipschitz_on_mult_const'*:
shows ‹∀ *x* ∈ *set xs* . |*c*|−*lipschitz_on* (*set xs*) (λ *y* . *c* ∗ *y*)›
⟨*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* ⇒ *nat* **where**
dim_vecf v = *CARD*('*n*)

definition *dim_colf* :: ('*a*, '*nc*::*finite*, '*nr*::*finite*) *fixed_mat* ⇒ *nat* **where**
dim_colf m = *CARD*('*nc*)

definition *dim_rowf* :: ('*a*, '*nc*::*finite*, '*nr*::*finite*) *fixed_mat* ⇒ *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* ⇒ ('*a*, '*nr*, '*nc*) *fixed_mat* ⇒ ('*a*, '*nr*, '*nc*) *fixed_mat* **where**
fixed_mat_add A B = *Abs_fixed_mat* (*Rep_fixed_mat A* + *Rep_fixed_mat B*)

definition *fixed_vec_zero* :: ('*a*::*zero*, '*nr*::*finite*) *fixed_vec* **where**

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

definition *fixed_vec_add* :: ('*a*::*plus*, '*nr*::*finite*) *fixed_vec* ⇒ ('*a*, '*nr*) *fixed_vec* ⇒ ('*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* ⇒ ('*a*, '*nr*::*finite*, '*nc*::*finite*) *fixed_mat* ⇒ ('*a*, '*nr*, '*nc*) *fixed_mat*
is λ*c A*. *c* ·$_m$ *A*
⟨*proof*⟩

lift_definition *fixed_mat_index* :: ('*a*, '*nr*::*finite*, '*nc*::*finite*) *fixed_mat* ⇒ *nat* ⇒ *nat* ⇒ '*a*
is λ*A i j*. *A* $$ (*i*, *j*) ⟨*proof*⟩

lift_definition *fixed_vec_index* :: ('*a*, '*nr*::*finite*) *fixed_vec* ⇒ *nat* ⇒ '*a*
is *vec_index* ⟨*proof*⟩

lift_definition *fixed_vec_smult* :: '*a*::*times* ⇒ ('*a*, '*nr*::*finite*) *fixed_vec* ⇒ ('*a*, '*nr*) *fixed_vec*
is λ*c A*. *c* ·$_v$ *A*
⟨*proof*⟩

lift_definition *mult_vec_fixed_mat* ::
('*a*::*semiring_0*, '*nr*::*finite*) *fixed_vec* ⇒ ('*a*, '*nr*, '*nc*::*finite*) *fixed_mat* ⇒ ('*a*, '*nc*) *fixed_vec*
(**infixl** $_{fv}$* 70)
is λ*v A*. *vec* (*dim_col A*) (λ*i*. *col A i* • *v*)
⟨*proof*⟩

lift_definition *map_fixed_vec* :: ('*a* ⇒ '*b*) ⇒ ('*a*, '*n*::*finite*) *fixed_vec* ⇒ ('*b*, '*n*::*finite*) *fixed_vec*
is *map_vec* :: ('*a* ⇒ '*b*) ⇒ '*a vec* ⇒ '*b vec*
⟨*proof*⟩

lemma *zero_in_carrier*:
o_m (*CARD*('*nr*)) (*CARD*('*nc*)) ∈ *carrier_mat* (*CARD*('*nr*)) (*CARD*('*nc*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *Rep_fixed_mat_add* [*simp*]:
Rep_fixed_mat (*fixed_mat_add A B*) = *Rep_fixed_mat A* + *Rep_fixed_mat B*
⟨*proof*⟩

lemma *Rep_fixed_vec_zero* [*simp*]:
Rep_fixed_vec (*fixed_vec_zero* :: ('*a*::*zero*, '*n*::*finite*) *fixed_vec*) = o_v (*CARD*('*n*))
⟨*proof*⟩

lemma *Rep_fixed_vec_add* [*simp*]:
Rep_fixed_vec (*fixed_vec_add A B*) = *Rep_fixed_vec A* + *Rep_fixed_vec B*
⟨*proof*⟩

lemma *Rep_fixed_mat_inject*: *Rep_fixed_mat A* = *Rep_fixed_mat B* ⟹ *A* = *B*
⟨*proof*⟩

lemma *Rep_fixed_vec_inject*: *Rep_fixed_vec A* = *Rep_fixed_vec B* ⟹ *A* = *B*
⟨*proof*⟩

lift_definition *row_fixed* :: $('a, 'n::finite, 'm::finite)$ *fixed_mat* $\Rightarrow$ *nat* $\Rightarrow$ $('a, 'm)$ *fixed_vec* **is**
$\lambda A\ i.\ vec\ (CARD('m))\ (\lambda j.\ A\ \$\$\ (i, j))$
$\langle proof \rangle$

lift_definition *col_fixed* :: $('a, 'n::finite, 'm::finite)$ *fixed_mat* $\Rightarrow$ *nat* $\Rightarrow$ $('a, 'n)$ *fixed_vec* **is**
$\lambda A\ j.\ vec\ (CARD('n))\ (\lambda i.\ A\ \$\$\ (i, j))$
$\langle proof \rangle$

lemma $CARD(285) = 285$ $\langle proof \rangle$

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**
$\lambda A\ B.\ mat\ (CARD('n))\ (CARD('k))\ (\lambda(i,j).$
sum_list $(map\ (\lambda l.\ A\ \$\$\ (i,l) * B\ \$\$\ (l,j))\ [0..<CARD('m)]))$
$\langle proof \rangle$

instance $\langle proof \rangle$

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 $=$ *sqrt* $(\sum i{\in}\{0..<CARD('nr)\}.\ \sum j{\in}\{0..<CARD('nc)\}.\ (norm\ (fixed\_mat\_index\ A\ i\ j))\hat{}2)$

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 $= (\forall x{\in}S.\ \forall_F\ (x', y)$ *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* :: $('a, 'nr::finite, 'nc::finite)$ *fixed_mat*) $+$ *A* $=$ *o*
⟨*proof*⟩

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

lemma *scaleR*: *1* $*_R$ *x* $=$(*x*:: $('a, 'nr::finite, 'nc::finite)$ *fixed_mat*)
⟨*proof*⟩

lemma *scaleR_o*: *o* $*_R$ *x* $=$(*o*:: $('a, 'nr::finite, 'nc::finite)$ *fixed_mat*)
⟨*proof*⟩

lemma *norm_o*: *norm* (*o*::$('a, 'nr::finite, 'nc::finite)$ *fixed_mat*) $=$ *o*
⟨*proof*⟩

lemma *sgn*: *sgn x* $=$ *inverse* (*norm x*) $*_R$ (*x*::$('a, 'nr::finite, 'nc::finite)$ *fixed_mat*)
⟨*proof*⟩

lemma *norm_eq_zero_iff*: (*norm x* $=$ (*o*::*real*)) $=$ (*x* $=$ (*o*::$('a, 'nr::finite, 'nc::finite)$ *fixed_mat*))
⟨*proof*⟩

lemma *sum_tuple*: ‹$(\sum\ i < n.\ \sum\ j < m\ .$ *P i j*$) = (\sum p{\in}\ \{(i,j).\ i < n \wedge j < m\}.$ *P* (*fst p*) (*snd p*))›
⟨*proof*⟩

lemma *triangle_inequality*: *norm* ((*x*::$('a, 'nr::finite, 'nc::finite)$ *fixed_mat*) $+$ *y*::$('a, 'nr::finite, 'nc::finite)$ *fixed_mat*)
$\leq$ *norm x* $+$ *norm y*
⟨*proof*⟩

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

instance
⟨*proof*⟩
end

instantiation *fixed_vec* :: $(\{real\_normed\_vector, times, one, real\_algebra\_1\}, finite)$ *real_normed_vector*
begin

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

⟨proof⟩

lift_definition *plus_fixed_vec* :: ('a, 'b) *fixed_vec* ⇒ ('a, 'b) *fixed_vec* ⇒ ('a, 'b) *fixed_vec* **is**
fixed_vec_add ⟨proof⟩

definition *scaleR_fixed_vec* :: *real* ⇒ ('a, 'b) *fixed_vec* ⇒ ('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* ⇒ ('a, 'b) *fixed_vec* **is**
λ *v*. *smult_vec* (−1) *v*
⟨proof⟩

lift_definition *minus_fixed_vec* :: ('a, 'b) *fixed_vec* ⇒ ('a, 'b) *fixed_vec* ⇒ ('a, 'b) *fixed_vec* **is**
λ *v w*. *v* + (*smult_vec* (−1) *w*)
⟨proof⟩

definition *norm_fixed_vec* :: ('a, 'b::*finite*) *fixed_vec* ⇒ *real* **where**
norm_fixed_vec A = *sqrt* (∑ *i*∈{0..<*CARD*('b)} . (*norm* (*fixed_vec_index A i*))^2)

definition *sgn_fixed_vec* :: ('a, 'b::*finite*) *fixed_vec* ⇒ ('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* ⇒ ('a, 'b) *fixed_vec* ⇒ *real* **where**
dist_fixed_vec v w = *norm* (*v* − *w*)

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

definition *open_fixed_vec* :: ('a, 'b) *fixed_vec set* ⇒ *bool* **where**
open_fixed_vec U = (∀ *x*∈*U*. ∀ *F* (*x'*, *y*) *in uniformity*. *x'* = *x* ⟶ *y* ∈ *U*)

lemma *uminus_add_vec*: − (*A* :: ('a, 'n::*finite*) *fixed_vec*) + *A* = 0
⟨proof⟩

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

lemma *scaleR_vec*: *1* *$_R$ *x* =(*x*:: ('a, 'n::*finite*) *fixed_vec*)
⟨proof⟩

lemma *norm_0_vec*: *norm* (0::('a, 'n::*finite*) *fixed_vec*) = 0
⟨proof⟩

lemma *scaleR_0_vec*: *0* *$_R$ *x* =(0:: ('a, 'n::*finite*) *fixed_vec*)
⟨proof⟩

lemma *sgn_vec*: *sgn x* = *inverse* (*norm x*) *$_R$ (*x*::('a, 'n::*finite*) *fixed_vec*)
⟨proof⟩

lemma *norm_eq_zero_iff_vec*: $(norm\ x = (o::real)) = (x = (o::('a, 'n::finite)\ fixed\_vec))$
⟨*proof*⟩

lemma *triangle_inequality_vec*: $norm\ ((x::('a, 'n::finite)\ fixed\_vec) + y::('a, 'n::finite)\ fixed\_vec) \leq norm\ x + norm\ y$
⟨*proof*⟩

lemma *norm_scaleR_vec*: $norm\ (a *_R x) = |a| * norm\ (x::('a, 'n::finite)\ fixed\_vec)$
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *uminus_fixed_vec*:
assumes $(v::'a::\{real\_algebra\_1,real\_normed\_vector\}\ Matrix.vec) \in carrier\_vec\ (CARD('n::finite))$
shows $-\ Abs\_fixed\_vec\ v = (Abs\_fixed\_vec\ (-\ v))::('a::\{real\_algebra\_1,real\_normed\_vector\}, 'n::finite)\ fixed\_vec)$
⟨*proof*⟩

lemma *lipschitz_on_mat_add*:
shows ‹$(1::real)-lipschitz\_on\ U\ (\lambda\ (A::('a::\{real\_algebra\_1,real\_normed\_vector\}, 'nr::finite, 'nc::finite)\ fixed\_mat)\ .\ A + M)$›
⟨*proof*⟩

lemma *vec_minus_element*:
fixes *v w* :: $'a::\{minus, zero\}\ vec$
assumes $dim\_vec\ v = dim\_vec\ w$ **and** $i < dim\_vec\ v$
shows $vec\_index\ (v - w)\ i = vec\_index\ v\ i - vec\_index\ w\ i$
⟨*proof*⟩

lemma *vec_minus*:
fixes *v w* :: $'a::\{minus, zero\}\ vec$
assumes $dim\_vec\ v = dim\_vec\ w$ **and** $i < dim\_vec\ v$
shows $(v - w) = vec\ (dim\_vec\ v)\ (\lambda i.\ vec\_index\ v\ i - vec\_index\ w\ i)$
⟨*proof*⟩

lemma *Rep_fixed_vec_plus*:
$Rep\_fixed\_vec\ ((u::('a::\{real\_algebra\_1,real\_normed\_vector\}, 'n::finite)\ fixed\_vec) +$
$(v::('a::\{real\_algebra\_1,real\_normed\_vector\}, 'n::finite)\ fixed\_vec)) =$
$Rep\_fixed\_vec\ u + Rep\_fixed\_vec\ v$
⟨*proof*⟩

lemma *fixed_vec_add*:
assumes $v1 \in carrier\_vec\ (CARD('n::finite))$
and $v2 \in carrier\_vec\ (CARD('n::finite))$
shows $Abs\_fixed\_vec\ v1 + Abs\_fixed\_vec\ v2 = (Abs\_fixed\_vec\ (v1 + v2))::('a::\{real\_algebra\_1,real\_normed\_vector\}, 'n)\ fixed\_vec)$
⟨*proof*⟩

lemma *col_minus_mat*:
fixes *A B* :: *'a*::{*minus*, *zero*} *mat*
assumes *dim_row A* = *dim_row B* **and** *dim_col A* = *dim_col B* **and** *i* < *dim_col A*
shows *col* (*A* − *B*) *i* = *col A i* − *col B i*
⟨*proof*⟩

lemma *index_vec_mat_mult*:
assumes *v* ∈ *carrier_vec* (*dim_row A*)
and *A* ∈ *carrier_mat* (*dim_row A*) (*dim_col A*)
and *i* < *dim_col* (*A*::*'a*::{*semiring_0*, *ab_semigroup_mult*} *Matrix.mat*)
shows (*v* $_v*$ *A*) $ *i* = (∑ *j* = *0*..<*dim_row A*. *v* $ *j* * *A* $$ (*j*, *i*))
⟨*proof*⟩

lemma *Rep_fixed_mat_minus*:
Rep_fixed_mat ((*x* − *y*)::(*'a*, *'b*, *'c*) *fixed_mat*) = *Rep_fixed_mat x* − *Rep_fixed_mat* (*y*:: (*'a*::{*real_algebra_1*,*real_normed_vector*}, *'b*::*finite*, *'c*::*finite*) *fixed_mat*)
⟨*proof*⟩

lemma *vector_matrix_inequality*:
fixes *c* :: (*'a*::{*real_normed_field*,*real_inner*}, *'nr*::*finite*) *fixed_vec*
and *U* :: (*'a*, *'nr*, *'nc*::*finite*) *fixed_mat set*
and *C* :: *real*
assumes *C_bound*: *C* ≥ *norm c*
and *C_nonneg*: *C* ≥ *0*
shows ⋀*x y*. *x* ∈ *U* ⟹ *y* ∈ *U* ⟹
sqrt (∑ *i* = *0*..<*CARD*(*'nc*). (*norm* (*fixed_vec_index* (*c* $_{fv}*$ *x* − *c* $_{fv}*$ *y*) *i*))^2) ≤
C * *sqrt* (∑ *i* = *0*..<*CARD*(*'nr*). ∑ *j* = *0*..<*CARD*(*'nc*). (*norm* (*fixed_mat_index* (*x* − *y*) *i j*))^2)
⟨*proof*⟩

lemma *lipschitz_on_mat_mult*:
assumes ‹*0* ≤ *C* › **and** *norm c* ≤ *C*
shows ‹*C*−*lipschitz_on U* (λ (*y*::(*'a*:: {*real_inner*,*real_normed_field*}, *'nr*::*finite*, *'nc*::*finite*) *fixed_mat*) .
(*c*::(*'a*, *'nr*) *fixed_vec*) $_{fv}*$ *y*)›
⟨*proof*⟩

lemma *lipschitz_on_weighted_sum_bias*:
assumes ‹*0* ≤ *C* › **and** *norm c* ≤ *C*
shows ‹*C*−*lipschitz_on U* (λ (*y*::(*'a*:: {*real_inner*,*real_normed_field*}, *'nr*::*finite*, *'nc*::*finite*) *fixed_mat*) . (*c* $_{fv}*$ *y*) + *b*)›
⟨*proof*⟩

lemma *mult_vec_mat_distrib_left*:
assumes *v1* ∈ *carrier_vec n* **and** *v2* ∈ *carrier_vec n* **and** *A* ∈ *carrier_mat n m*
shows (*v1* − *v2*) $_v*$ *A* = *v1* $_v*$ *A* − *v2* $_v*$ (*A*:: *'a*::{*real_normed_field*,*real_inner*} *mat*)
⟨*proof*⟩

lemma *matrix_vector_inequality*:
fixes *c* :: (*'a*::{*real_normed_field*,*real_inner*}, *'nr*::*finite*, *'nc*::*finite*) *fixed_mat*
and *U* :: (*'a*, *'nr*) *fixed_vec set*
and *C* :: *real*
assumes *C_bound*: *C* ≥ *norm c*
and *C_nonneg*: *C* ≥ *0*
shows ⋀*x y*. *x* ∈ *U* ⟹ *y* ∈ *U* ⟹
sqrt (∑ *i* = *0*..<*CARD*(*'nc*). (*norm* (*fixed_vec_index* (*x* $_{fv}*$ *c* − *y* $_{fv}*$ *c*) *i*))^2) ≤

$C * sqrt\ (\sum i = 0..<CARD('nr).\ (norm\ (fixed\_vec\_index\ (x - y)\ i))^2)$
⟨proof⟩

lemma *lipschitz_on_vec_mult*:
assumes ‹$0 \leq C$› **and** *norm c* $\leq C$
shows ‹*C*−*lipschitz_on U* (λ *y* . *y* $_{fv}*$ (*c*::('*a*:: {*real_inner*,*real_normed_field*}, '*nr*::*finite*, '*nc*::*finite*) *fixed_mat*))›
⟨proof⟩

lemma *C_ge_norm*:
norm (*c*::('*a*:: {*real_algebra_1*,*real_normed_vector*}, '*nr*::*finite*, '*nc*::*finite*) *fixed_mat*) $\leq C \Longrightarrow 0 \leq C$
⟨proof⟩

lemma *lipschitz_on_weighted_sum_bias'*:
assumes *norm c* $\leq C$
shows ‹*C*−*lipschitz_on U* (λ *y* . (*y* $_{fv}*$ (*c*::('*a*:: {*real_inner*,*real_normed_field*}, '*nr*::*finite*, '*nc*::*finite*) *fixed_mat*)) + *b*)›
⟨proof⟩

lemma *lipschitz_on_dense*:
assumes *norm c* $\leq C$
assumes *D*−*lipschitz_on* ((λ*y*. (*c*::('*a*:: {*real_inner*,*real_normed_field*}, '*n*::*finite*) *fixed_vec*) $_{fv}*$ *y* + *b*) ' *U*) *f*
shows (*D* * *C*)−*lipschitz_on U* (λ *y* . *f* ((*c* $_{fv}*$ *y*) + *b*))
⟨proof⟩

lemma *lipschitz_on_dense'*:
assumes *norm c* $\leq C$
assumes *D*−*lipschitz_on* ((λ*y*. *y* $_{fv}*$ (*c*::('*a*:: {*real_inner*,*real_normed_field*}, '*nr*::*finite*, '*nc*::*finite*) *fixed_mat*) + *b*) ' *U*) *f*
shows (*D* * *C*)−*lipschitz_on U* (λ *y* . *f* ((*y* $_{fv}*$ *c*) + *b*))
⟨proof⟩

lemma *lipschitz_on_input_ouput*:
shows *1*−*lipschitz_on U* (λ *i* . *i*)
⟨proof⟩

lemma *lipschhitz_on_activation*:
assumes *C*−*lipschitz_on U f*
shows *C*−*lipschitz_on U* (λ *i* . *f i*)
⟨proof⟩

Neural Network: Layers **fun** $predict_{layer\_f}$ :: (('*a*::{*real_algebra_1*,*real_normed_vector*}, '*b*::*finite*) *fixed_vec*, '*c*, '*d*) *neural_network_seq_layers*
⇒ ('*a*, '*b*::*finite*) *fixed_vec* ⇒ (('*a*, '*b*) *fixed_vec*, '*c*, ('*a*, '*b*, '*b*) *fixed_mat*) *layer* ⇒ ('*a*, '*b*) *fixed_vec*
where
‹$predict_{layer\_f}$ *N* (*vs*) (*In l*) = *vs* ›
| ‹$predict_{layer\_f}$ *N* (*vs*) (*Out l*) = *vs*›
| ‹$predict_{layer\_f}$ *N* (*vs*) (*Dense pl*) = ((*the* (*activation_tab N* (φ *pl*))) ((*vs* $_{fv}*$ ω *pl*) + β *pl*)) ›
| ‹$predict_{layer\_f}$ *N* (*vs*) (*Activation pl*) = ((*the* (*activation_tab N* (φ *pl*))) *vs*) ›

definition ‹$predict_{seq\_layer\_f}$ *N inputs* = *foldl* ($predict_{layer\_f}$ *N*) *inputs* (*layers N*)›

definition ‹*lipschitz_continuous_activation_tab*$_f$ *tab U* = (∀ *f* ∈ *ran* (*tab*). ∃ *C*. *C*−*lipschitz_on U f*)›

lemma *input_layer_lipschitz_on*:
1−*lipschitz_on U* ((λ *vs* . (*predict*$_{layer\_f}$ *N vs* (*In x1*))))
⟨*proof*⟩

lemma *output_layer_lipschitz_on*:
1−*lipschitz_on U* ((λ *vs* . (*predict*$_{layer\_f}$ *N vs* (*Out x1*))))
⟨*proof*⟩

lemma *dense_layer_lipschitz_on*:
assumes *norm* (ω *x1*) ≤ *C*
assumes *D*−*lipschitz_on* ((λ*y*. *y* $_{fv}$* ω *x1* + β *x1*) ' *U*) (*the* (*activation_tab N* (φ *x1*)))
shows (*C* ∗ *D*)−*lipschitz_on U* (λ *vs*::('*a*:: {*real_inner*,*real_normed_field*}, '*c*::*finite*) *fixed_vec* . (*predict*$_{layer\_f}$ *N vs* (*Dense x1*)))
⟨*proof*⟩

lemma *activation_layer_lipschitz_on*:
assumes *C*−*lipschitz_on U* (*the* (*activation_tab N* (φ *x1*)))
shows *C*−*lipschitz_on U* (λ *vs* . (*predict*$_{layer\_f}$ *N vs* (*Activation x1*)))
⟨*proof*⟩

lemma *foldl_layer_lipschitz_on*:
fixes *N* :: (('*a*::{*real_algebra_1*,*real_normed_vector*}, '*b*::*finite*) *fixed_vec*, '*c*, '*d*) *neural_network_seq_layers*
assumes *layer_lipschitz*: ∀ *l* ∈ *set ls*. ∃ *C*. *C*−*lipschitz_on U* (λ*vs*. *predict*$_{layer\_f}$ *N vs l*)
assumes *domain_invariant*: ∀ *l* ∈ *set ls*. ∀ *vs* ∈ *U*. *predict*$_{layer\_f}$ *N vs l* ∈ *U*
shows ∃ *C*. *C*−*lipschitz_on U* (λ*vs*. *foldl* (*predict*$_{layer\_f}$ *N*) *vs ls*)
⟨*proof*⟩

lemma *layers_lipschitz_from_components*:
fixes *N* :: (('*a*::{*real_algebra_1*,*real_normed_vector*,*real_inner*,*real_normed_field*}, '*b*::*finite*) *fixed_vec*, '*c*, '*d*) *neural_network_seq_layers*
assumes *weight_bounded*: ∀ *l* ∈ *set ls*. (*case l of Dense pl* ⇒ *norm* (ω *pl*) ≤ *W*)
assumes *activation_lipschitz*: ∀ *l* ∈ *set ls*. (*case l of*
Dense pl ⇒ ∃ *D*. *D*−*lipschitz_on* ((λ*y*. *y* $_{fv}$* ω *pl* + β *pl*) ' *U*) (*the* (*activation_tab N* (φ *pl*))) |
Activation pl ⇒ ∃ *C*. *C*−*lipschitz_on U* (*the* (*activation_tab N* (φ *pl*))))
shows ∀ *l* ∈ *set ls*. ∃ *C*. *C*−*lipschitz_on U* (λ*vs*. *predict*$_{layer\_f}$ *N vs l*)
⟨*proof*⟩

lemma *Rep_fixed_vec_minus*: *Rep_fixed_vec* (*x* − *y*) = *Rep_fixed_vec x* − *Rep_fixed_vec y*
⟨*proof*⟩

Lipschitz Continuity of Functions (Interval)

Neural Network: Activations **lemma** *relu_lipschitz'*: ⋀*x y*. (*x*::(*real*, '*b*::*finite*) *fixed_vec*) ∈ *X* ⟹
(*y*::(*real*, '*b*::*finite*) *fixed_vec*) ∈ *X* ⟹
dist
((*Abs_fixed_vec*
(*Matrix.vec* (*dim_vec* (*Rep_fixed_vec x*))
(λ*i*. *if* 0 ≤ *Rep_fixed_vec x* $ *i then Rep_fixed_vec x* $ *i else* 0)))::(*real*, '*b*::*finite*) *fixed_vec*)
((*Abs_fixed_vec*
(*Matrix.vec* (*dim_vec* (*Rep_fixed_vec y*))
(λ*i*. *if* 0 ≤ *Rep_fixed_vec y* $ *i then Rep_fixed_vec y* $ *i else* 0)))::(*real*, '*b*::*finite*) *fixed_vec*)
≤ *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': ⋀x y. (x::(real, 'b::finite) fixed_vec) ∈ X ⟹
(y::(real, 'b::finite) fixed_vec) ∈ X ⟹
dist ((map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (λf v. Matrix.vec (dim_vec v) (λi. f (v $ i)))) softplus x)::(real, 'b) fixed_vec)
(map_fun id (map_fun Rep_fixed_vec Abs_fixed_vec) (λf v. Matrix.vec (dim_vec v) (λi. f (v $ i)))) softplus y)
≤ 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 ⇒ 'b list) ⇒ ('a list ⇒ 'b list) ⇒ bool (_ ≡$_{l}$ _)
where
‹layer_equiv f g = (∃ p p'. ∀ 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 ≡$_{l}$ 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›

| ‹input_neuron (Neuron n) = False›

fun *hidden_neuron* **where**
‹hidden_neuron (In nid) = False›
| ‹hidden_neuron (Out nid) = False›
| ‹hidden_neuron (Neuron n) = True›

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* :: ‹('a::{one}, 'b, 'c) neural_network ⇒ 'a ⇒ 'b ⇒ 'a ⇒ 'a ⇒ id ⇒ id ⇒ ('a, 'b) edge›
where
‹mk_edge N ω' φ' α' β' id' nid = (| ω = ω',
tl = (the_elem {n . n ∈ neurons N ∧ uid n = id'}),
hd = Neuron (| φ = φ', α = α', β = β', uid = nid |) |) ›

definition *mk_out_edge* :: ‹('a::{one}, 'b, 'c) neural_network ⇒ id ⇒ id ⇒ ('a, 'b) edge›
where
‹mk_out_edge N id' nid' = (| ω = 1,
tl = (the_elem {n . n ∈ neurons N ∧ uid n = id'}),
hd = Out nid' |)›

definition *mk_new_ids* :: ‹('a::{one}, 'b, 'c) neural_network ⇒ nat list ›
where
‹mk_new_ids N = upt (Max(uids (graph N)) + 1)
(Max(uids (graph N)) + card (output_layer_ids N) + 1)›

mk_new_ids makes a list of new ids corresponding to the size of the current last layer in a given network and the current maximum id in the network. This is used in the activation and out functions in order to generate the new neurons in the edges. In order to help validate that the *mk_new_ids* returns the correct sized list and that the ids are unique in the network the following lemmas are needed to simplify this.

lemma *new_id_len*: ‹length(mk_new_ids N) = length(sorted_list_of_set(output_layer_ids N))›
⟨*proof*⟩

lemma *new_id_len_card*: ‹length(mk_new_ids N) = card(output_layer_ids N)›
⟨*proof*⟩

lemma *new_id_distinct*: ‹distinct(mk_new_ids N)›
⟨*proof*⟩

lemma *new_id_greater*:
assumes ‹card (output_layer_ids N) > 0›
shows ‹Min(set(mk_new_ids N)) > Max(uids (graph N))›
⟨*proof*⟩

lemma *new_id_sorted*:
shows ‹sorted (mk_new_ids N)›
⟨*proof*⟩

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
 ‹layers_digraph _ N (⦇ω=_, tl=_, hd=In _⦈)      = (o, ERROR)›
| ‹layers_digraph _ N (⦇ω=_, tl=Out _ , hd=_ ⦈)  = (o, ERROR)›
| ‹layers_digraph _ N (⦇ω=_, tl=In uid_in , hd=_ ⦈)  = (o, OK)›
| ‹layers_digraph 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 layers_digraph (n−1) N e' of
                                  (_, ERROR) ⇒ ((o,o), ERROR)
                                | (v, OK)    ⇒ ((v+1 ,uid (tl e')), OK))) ' E')
              in
              (Max ((λ a .fst(fst a )) ' {n. n ∈ lvals ∧ snd n = OK }) , OK))
            else (o, ERROR))›
```

Layers are defined as the path from the output node, this allows all dependencies to be calculated before prediction. In $layers_{digraph}$ the layer is calculated using the edges.

```
fun layers_digraph_neuron :: ‹nat ⇒ ('a::{zero,linorder,numeral}, 'b, 'c) neural_network
          ⇒ ('a, 'b) neuron ⇒ ('a × error)›
 where
 ‹layers_digraph_neuron _ N (In uid_in) = (o, OK)›
| ‹layers_digraph_neuron n N (Out uid_out) = (if o < n then
                        (let
                          E' = tl ' (incoming_arcs N uid_out);
                          lvals = ((λ n' . (case layers_digraph_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))›
| ‹layers_digraph_neuron n N (Neuron a) = (if o < n then
                        (let
                          E' = tl ' (incoming_arcs N (Neuron.uid a));
                          lvals = ((λ n' . (case layers_digraph_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 $layers_{digraph\_neuron}$ 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 layers_edges :: ‹'a ⇒ 'a ⇒ ('a::{zero,numeral,linorder}, 'b, 'c) neural_network
          ⇒ ('a, 'b) edge set› where
 ‹layers_edges l l' N = (let n_all  = neurons N;
                layer =  (λ n . ((layers_digraph_neuron (card n_all) N n), uid n)) ' n_all;
                n_in   = snd ' {n . n ∈ layer ∧ fst(fst n) = l};
                n_out  = snd ' {n . n ∈ layer ∧ fst(fst n) = l'}
               in { e . e ∈ edges N ∧ uid (tl e) ∈ n_in ∧ uid (hd e) ∈ n_out } )›
```

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 $isDense_s :: ‹('a, 'b)\ edge\ set \Rightarrow bool›$ **where**
$‹isDense_s\ e = ((\forall\ n' \in\ tl\ `\ e\ .\ \forall\ n'' \in hd\ `\ e\ .\ \exists\ e' \in e\ .\ tl\ e' = n' \wedge hd\ e' = n''))›$

definition $isActivation_s :: ‹('a, 'b)\ edge\ set \Rightarrow bool›$ **where**
$‹isActivation_s\ e = ((\forall\ n' \in\ tl\ `\ e\ .\ \exists!\ e' \in e\ .\ tl\ e' = n') \wedge$
$(\forall\ n'' \in hd\ `\ e\ .\ \exists!\ e'' \in e\ .\ hd\ e'' = n''))›$

definition $isInput_s :: ‹('a, 'b)\ edge\ set \Rightarrow bool›$ **where**
$‹isInput_s\ e = (isDense_s\ e \wedge (\forall\ n \in hd\ `\ e\ .\ input\_neuron\ n))›$

definition $isOutput_s :: ‹('a, 'b)\ edge\ set \Rightarrow bool›$ **where**
$‹isOutput_s\ e = (isActivation_s\ e \wedge (\forall\ n''' \in hd\ `\ e\ .\ output\_neuron\ n'''))›$

The following for functions test whether lists of edges correspond to the correct type of connections for Dense, Activation, Input and Output layers. We want these definitions over lists and sets in order to allow us to use whichever is more efficient in specific situations.

definition $isDense_l :: ‹('a, 'b)\ edge\ list \Rightarrow bool›$ **where**
$‹isDense_l\ e = (let\ t = (map\ tl\ e);\ h = (map\ hd\ e)\ in$
$(\forall\ n' \in\ set\ t\ .\ \forall\ n'' \in set\ h\ .$
$filter\ (\lambda\ e'\ .\ tl\ e' = n' \wedge hd\ e' = n'')\ e\ \neq [])›$

definition $isInput_l :: ‹('a, 'b)\ edge\ list \Rightarrow bool›$ **where**
$‹isInput_l\ e = (isDense_l\ e\ \wedge foldr\ (\wedge)\ (map\ input\_neuron\ (map\ hd\ e))\ True)›$

definition $isActivation_l :: ‹('a, 'b)\ edge\ list \Rightarrow bool›$ **where**
$‹isActivation_l\ e = (let\ t = (map\ tl\ e);\ h = (map\ hd\ e)\ in$
$distinct\ t \wedge distinct\ h \wedge length\ t = length\ h \wedge length\ e = length\ h \wedge$
$length\ t = length\ e\ )›$

definition $isOutput_l :: ‹('a, 'b)\ edge\ list \Rightarrow bool›$ **where**
$‹isOutput_l\ e = (isActivation_l\ e \wedge foldr\ (\wedge)\ (map\ (output\_neuron \circ hd)\ e)\ True)›$

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 $‹foldr\ (\wedge)\ (map\ (output\_neuron \circ hd)\ e)\ True = \ (\forall\ n' \in hd\ `\ set\ e\ .\ output\_neuron\ n')›$
$\langle proof \rangle$

lemma *allInput*:
shows $‹foldr\ (\wedge)\ (map\ (input\_neuron \circ hd)\ e)\ True = \ (\forall\ n' \in hd\ `\ set\ e\ .\ input\_neuron\ n')›$
$\langle proof \rangle$

lemma *forAll*:
$‹(\forall\ n' \in set\ (map\ tl\ e).\forall\ n'' \in set(map\ hd\ e).filter(\lambda e'.\ tl\ e' = n' \wedge hd\ e' = n'')\ e \neq []) =$
$(\forall\ n' \in tl`\ set\ e.\ \forall\ n'' \in hd\ `\ set\ e\ .\ \exists\ e' \in set\ e\ .\ tl\ e' = n' \wedge hd\ e' = n'')›$
$\langle proof \rangle$

lemma $isDense_l\_isDense_s\_equivalence$: $‹isDense_l\ E = isDense_s\ (set\ E)›$
$\langle proof \rangle$

lemma $isnput_l\_isInput_s\_equivalence$: $‹isInput_l\ E = isInput_s\ (set\ E)›$

⟨*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* (λ*x*. *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 ‹*length*(*mk_new_ids N*) = *length*(*activation N* φ' α' β')›
⟨*proof*⟩

lemma *map_neuron_hd_id*:
‹(*map* (λ*x*. *Neuron* (|φ = φ', α = α', β = β', *uid* = *f x*|)) *X*) =
(*map* (λ*x*. *Neuron* (|φ = φ', α = α', β = β', *uid* = *x*|)) (*map f X*))›
⟨*proof*⟩

lemma *map_neuron_tl_id*:
‹(*map* (λ*x*. *the_elem* {*n* ∈ *neurons N*. *NN_Digraph.uid n* = *f x*}) *X*) =
(*map* (λ*x*. *the_elem* {*n* ∈ *neurons N*. *NN_Digraph.uid n* = *x*}) (*map f X*))›
⟨*proof*⟩

context *nn_pregraph* **begin**

lemma *distinct_head_activation*: ‹*distinct*(*map hd* (*activation N* φ' α' β'))›
⟨*proof*⟩

end

context *neural_network_digraph* **begin**

lemma *distinct_tail_activation*: ‹*distinct*(*map tl* (*activation N* φ' α' β'))›
⟨*proof*⟩

lemma *activation_is_activation*: ‹*isActivation*$_l$(*activation N* φ' α' β')›
⟨*proof*⟩

end

Output layer proofs **lemma** *output_output_layer_length_equal*:
assumes *notEmptyNeurons*: ‹*neurons N* ≠ {}›
and *notEmptyOutputLayer*: ‹*length*(*out N*) ≠ *o*›
shows ‹*card*(*output_layer_ids N*) = *length*(*out N*)›
⟨*proof*⟩

lemma *new_ids_output_layer_length_equal*:
assumes *notEmptyNeurons*: ‹*neurons N* ≠ {}›
and *notEmptyOutputLayer*: ‹*length*(*out N*) ≠ *o*›
shows ‹*length*(*mk_new_ids N*) = *length*(*out N*)›
⟨*proof*⟩

lemma *distinct_output_edges*: ‹*distinct* (*out N*)›
⟨*proof*⟩

lemma *map_out_neuron_hd_id*: ‹(*map* (λ*x*. *Out* (*f x*)) *X*) = (*map* (λ*x*. *Out x*) (*map f X*))›
⟨*proof*⟩

context *nn_pregraph* **begin**

lemma *distinct_head_output*: ‹*distinct*(*map hd* (*out N*))›
⟨*proof*⟩

end

lemma *fold_and_map*: ‹*foldr* (∧) (*map* (λ*x*. *True*) *X*) *True*›
⟨*proof*⟩

lemma *head_output_neurons*: ‹*foldr* (∧) (*map* (*output_neuron* ∘ *edge.hd*) (*out N*)) *True*›
⟨*proof*⟩

context *neural_network_digraph* **begin**

lemma *distinct_tail_output*: ‹*distinct*(*map tl* (*out N*))›
⟨*proof*⟩

lemma *output_is_output*:‹*isOutput*$_l$(*out N*)›
⟨*proof*⟩

Dense layer proofs **lemma** *dense_is_dense*:
assumes *neuronsNotZero*: ‹*n* > *o*›
and *weightEqualNeurons*: ‹*length* ω′ = *n*›
shows ‹*isDense*$_s$(*set*(*dense N n* ω′ φ′ α′ β′))›
⟨*proof*⟩

end
end

5.2.2 Neural Network as Sequential Layers using Lists (NN_Layers_List_Main)

theory
NN_Layers_List_Main
imports
Main
NN_Layers
HOL−Library.Interval
Properties
begin

definition ‹*valid_activation_tab*$_l$ *tab* = (∀ *f* ∈ *ran tab*. ∀ *xs*. *length xs* = *length* (*f xs*))›

lemma *valid_activation_preserves_length*:
assumes ‹*valid_activation_tab*$_l$ *t*›
assumes ‹*t n* = *Some f*›
shows ‹*length xs* = *length* (*f xs*)›
⟨*proof*⟩

fun *layer_consistent*$_l$ :: ('*a list*, '*b*, '*a list list*) *neural_network_seq_layers* ⇒ *nat* ⇒ ('*a list*, '*b* , '*a list list*) *layer* ⇒ *bool*
where
‹*layer_consistent*$_l$ _ *nc* (*In l*) = (*o* < *units l* ∧ *nc* = *units l*)›
| ‹*layer_consistent*$_l$ _ *nc* (*Out l*) = (*o* < *units l* ∧ *nc* = *units l*)›
| ‹*layer_consistent*$_l$ *N nc* (*Activation l*) = ((*o* < *units l* ∧ *nc* = *units l*)
∧ (((*activation_tab N*) (φ *l*)) ≠ *None*))›
| ‹*layer_consistent*$_l$ *N nc* (*Dense l*) = (*o* < *units l* ∧ *o* < *nc*
∧ *length* (β *l*) = *units l*
∧ *length* (ω *l*) = *units l*
∧ (∀ *r*∈*set* (ω *l*). *length r* = *nc*)

$\wedge$ (((*activation_tab N*) (*φ l*)) $\neq$ *None*))›

fun *layers_consistent$_l$* **where**
‹*layers_consistent$_l$ N _ [] = True* ›
|‹*layers_consistent$_l$ N w* (*l*#*ls*) = ((*layer_consistent$_l$ N w l*) ∧ (*layers_consistent$_l$ N* (*out_deg_layer l*) *ls*))›

lemma *layer_consistent$_l$_in_deg_layer*:
assumes *layer_consistent$_l$ N nc l*
shows *in_deg_layer l = nc*
⟨*proof*⟩

lemma *layers_consistent$_l$_in_deg*:
assumes (*layers_consistent$_l$ N nc* (*l*#*ls'*))
shows *in_deg_layer l = nc*
⟨*proof*⟩

lemma *layer_consistent$_l$_activation_tab_const*:
‹*layer_consistent$_l$ N nc l = layer_consistent$_l$* (|*layers = ls*, *activation_tab = activation_tab N*|) *nc l*›
⟨*proof*⟩

lemma *layers_consistent$_l$_activation_tab_const*:
‹*layers_consistent$_l$ N nc ls = layers_consistent$_l$* (|*layers = ls'*, *activation_tab = activation_tab N*|) *nc ls*›
⟨*proof*⟩

lemma *layers_consistent$_l$_layersN_const*:
‹*layers_consistent$_l$ N = layers_consistent$_l$* (|*layers = ls'*, *activation_tab = activation_tab N*|)›
⟨*proof*⟩

lemma *layers_consistent$_l$All*:
assumes ‹*layers_consistent$_l$ N inputs* (*layers N*)›
shows ‹∀ *l* ∈ *set* (*layers N*). ∃ *n* . *layer_consistent$_l$ N n l*›
⟨*proof*⟩

lemma *layers_consistent$_l$All'*:
assumes ‹*layers_consistent$_l$ N* (*in_deg_NN N*) (*layers N*)›
shows ‹∀ *l* ∈ *set* (*layers N*). ∃ *n* . *layer_consistent$_l$ N n l*›
⟨*proof*⟩

lemma *layers_consistent$_l$_layer_consistent$_l$_Dense*:
assumes ‹*layers_consistent$_l$ N* (*in_deg_NN N*) (*layers N*)›
and (*Dense x3*) ∈ *set* (*layers N*)
shows ‹*layer_consistent$_l$ N* (*length* (*ω x3* ! *o*)) (*Dense x3*) ›
⟨*proof*⟩

lemma *layers_consistent$_l$_layer_consistent$_l$_Activation*:
assumes ‹*layers_consistent$_l$ N* (*in_deg_NN N*) (*layers N*)›
and (*Activation x3*) ∈ *set* (*layers N*)
shows ‹*layer_consistent$_l$ N* (*units x3*) (*Activation x3*) ›
⟨*proof*⟩

```
locale neural_network_sequential_layers_l =
 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_tab_l (activation_tab N)›
   and layer_valid: ‹layers_consistent_l 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 predict_layer_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
   ‹predict_layer_l N (Some vs) (In l) = (if layer_consistent_l N (length vs) (In l) then Some vs else None)›
 | ‹predict_layer_l N (Some vs) (Out l) = (if layer_consistent_l N (length vs) (Out l) then Some vs else None)›
 | ‹predict_layer_l N (Some vs) (Dense pl) = (if layer_consistent_l 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)
                              ›
 | ‹predict_layer_l N (Some vs) (Activation pl) = (if layer_consistent_l N (length vs) (Activation pl) then
                                  (case activation_tab N (φ pl) of
                                    None ⇒ None
                                  | Some f ⇒ Some (f vs))
                                 else None)
                                 ›
 | ‹predict_layer_l _ None _ = None›
lemma length_out: ‹predict_layer_l N' vs (Out l) = Some res ⟹ length(res) = (units l)›
 ⟨proof⟩

fun
 predict_layer_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
‹*predict*$_{layer\_l}$*_impl N vs* (*In l*) = *vs*›
| ‹*predict*$_{layer\_l}$*_impl N vs* (*Out l*) = *vs*›
| ‹*predict*$_{layer\_l}$*_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*));
φ$_l$ = *the* (*activation_tab N* (φ *pl*))
in
φ$_l$ *wsum_bias*)
›

| ‹ *predict*$_{layer\_l}$*_impl N vs* (*Activation pl*) = (*let*
φ$_l$ = *the* (*activation_tab N* (φ *pl*))
in
φ$_l$ *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_consistent*$_l$ *N* (*length xs*) *l*)›
shows ‹(*predict*$_{layer\_l}$ *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**
‹*predict*$_{layer\_l}$ *N x xs* = *predict*$_{layer\_l}$ *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 ‹*predict*$_{layer\_l}$ *N* (*Some i*) (*In* (|*name* = *x*, *units* = *y*|)) = (*Some i*)›
⟨*proof*⟩

lemma *output_layer*:
assumes ‹$y = length\ i$› **and** ‹$o < y$›
shows ‹$predict_{layer\_l}\ N\ (Some\ i)\ (Out\ (\!|name = x, units = y|\!)) = (Some\ i)$›
⟨*proof*⟩

lemma *dense_layer*:
shows ‹$predict_{layer\_l}\ N\ (Some\ i)\ (Dense\ (\!|name = x, units = y, ActivationRecord.\varphi = p, LayerRecord.\beta = b, \omega = w|\!))$
$=$
$(if\ layer\_consistent_l\ N\ (length\ i)\ (Dense\ (\!|name = x, units = y, ActivationRecord.\varphi = p, LayerRecord.\beta = b, \omega = w|\!))\ then$
$(let\ in\_w\_pairs = map\ (\lambda\ e.\ zip\ i\ e)\ w;$
$wsums\ = map\ (\lambda\ vs'.\ \sum (x,y)\leftarrow vs'.\ x*y)\ in\_w\_pairs;$
$wsum\_bias\ = map\ (\lambda\ (s,b).\ s+b)\ (zip\ wsums\ b)$
in
$(case\ activation\_tab\ N\ p\ of$
$None\ \Rightarrow None$
$|\ Some\ f \Rightarrow Some\ (f\ wsum\_bias\ )))$
$else\ None)$›
⟨*proof*⟩

lemma *dense_layer′*:
assumes ‹$activation\_tab\ N\ p = Some\ a$›
shows ‹$predict_{layer\_l}\ N\ (Some\ i)\ (Dense\ (\!|name = x, units = y, ActivationRecord.\varphi = p, LayerRecord.\beta = b, \omega = w|\!))$
$=$
$(if\ layer\_consistent_l\ N\ (length\ i)\ (Dense\ (\!|name = x, units = y, ActivationRecord.\varphi = p, LayerRecord.\beta = b, \omega = w|\!))\ then$
$(let\ in\_w\_pairs = map\ (\lambda\ e.\ zip\ i\ e)\ w;$
$wsums\ = map\ (\lambda\ vs'.\ \sum (x,y)\leftarrow vs'.\ x*y)\ in\_w\_pairs;$
$wsum\_bias\ = map\ (\lambda\ (s,b).\ s+b)\ (zip\ wsums\ b)$
$in\ Some\ (a\ wsum\_bias))$
$else\ None)$›
⟨*proof*⟩

lemma *activation_layer*:
assumes ‹$y = length\ i$›
shows ‹$predict_{layer\_l}\ N\ (Some\ i)\ (Activation\ (\!|name = x, units = y, ActivationRecord.\varphi = p|\!)) =$
$(if\ layer\_consistent_l\ N\ (length\ i)\ (Activation\ (\!|name = x, units = y, ActivationRecord.\varphi = p|\!))\ then$
$(case\ activation\_tab\ N\ p\ of\ None \Rightarrow None\ |\ Some\ f \Rightarrow Some\ (f\ i))$
$else\ None)$›
⟨*proof*⟩

lemma *activation_layer′*:
assumes ‹$y = length\ i$›
and ‹$activation\_tab\ N\ p = Some\ a$›
shows ‹$predict_{layer\_l}\ N\ (Some\ i)\ (Activation\ (\!|name = x, units = y, ActivationRecord.\varphi = p|\!)) =$
$(if\ layer\_consistent_l\ N\ (length\ i)\ (Activation\ (\!|name = x, units = y, ActivationRecord.\varphi = p|\!))\ then\ Some\ (a\ i)\ else\ None)$›
⟨*proof*⟩

lemma $predict_{layer\_l}$*_impl_activation_tab_const*: ‹$predict_{layer\_l}$*_impl* $N = predict_{layer\_l}$*_impl* $(\!|layers = l, activation\_tab = activation\_tab\ N|\!)$›
⟨*proof*⟩

context *neural_network_sequential_layers*$_l$ **begin**

lemma *img_None_1*: **assumes** ‹((*predict*$_{seq\_layer\_l}$ *N xs*) ≠ *None*)› **shows** ‹(*length xs* = (*in_deg_NN N*))›
⟨*proof*⟩

lemma *img_None_2'*:
assumes *a0*: ‹*layers'* ≠ [] ›
and *a4*: ‹*valid_activation_tab*$_l$ *activation_tab'*›
and *a1*: ‹*layers_consistent*$_l$ (⦇*layers* = [], *activation_tab*=*activation_tab'* ⦈) (*length xs*) *layers'*›
shows ‹*foldl* (*predict*$_{layer\_l}$ (⦇*layers* = [], *activation_tab* = *activation_tab'*⦈)) (*Some xs*) *layers'* ≠ *None*›
⟨*proof*⟩

lemma *img_None_2*:
assumes ‹*length xs* = *in_deg_NN N*›
shows ‹((*predict*$_{seq\_layer\_l}$ *N xs*) ≠ *None*)›
⟨*proof*⟩

lemma *img_None*: ‹((*predict*$_{seq\_layer\_l}$ *N xs*) ≠ *None*) = (*length xs* = *in_deg_NN N*)›
⟨*proof*⟩

lemma *img_Some*: ‹(∃ *y*. (*predict*$_{seq\_layer\_l}$ *N xs*) = *Some y*) = (*length xs* = *in_deg_NN N*)›
⟨*proof*⟩

lemma *img_length*: ‹(∃ *y*. ((*predict*$_{seq\_layer\_l}$ *N xs*) = *Some y*) ⟶ (*length y* = *out_deg_NN N*))›
⟨*proof*⟩

lemma *predit*$_{layer\_l}$*_impl_eq*:
assumes ‹*layer_consistent*$_l$ *N* (*length inputs*) *l*›
shows ‹*predict*$_{layer\_l}$ *N* (*Some inputs*) *l* = *Some* (*predict*$_{layer\_l}$*_impl N inputs l*)›
⟨*proof*⟩

lemma *aux_length*: ‹
0 < *units x3* ⟹ *valid_activation_tab*$_l$ *atab* ⟹
inputs ≠ [] ⟹
length (*β x3*) = *units x3* ⟹
length (*ω x3*) = *units x3* ⟹
∀ *r*∈*set* (*ω x3*). *length r* = *length inputs* ⟹
atab (*φ x3*) = *Some y* ⟹
(*length* (*y* (*map2* (+) (*map* ((λ*vs'*. ∑ (*x*, *y*)←*vs'*. *x* ∗ *y*) ∘ *zip inputs*) (*ω x3*)) (*β x3*)))) = *units x3*
›
⟨*proof*⟩

lemma *pred_list_impl_aux'*:
assumes ‹*ls* ≠ []›
and *layer_valid*: ‹*layers_consistent*$_l$ (⦇*layers* = [], *activation_tab* = *atab*⦈) (*length inputs*) *ls*›
and *activation_tab_valid*: ‹*valid_activation_tab*$_l$ *atab*›
shows ‹
foldl (*predict*$_{layer\_l}$ (⦇*layers* = [], *activation_tab* = *atab*⦈)) (*Some inputs*) *ls* =
Some (*foldl* (*predict*$_{layer\_l}$*_impl* (⦇*layers* = [], *activation_tab* = *atab*⦈)) *inputs ls*)
›
⟨*proof*⟩

lemma *pred_list_impl_aux*:
assumes *layer_valid*: ‹*layers_consistent*$_l$ (|*layers* = *ls*, *activation_tab* = *atab*|) (*length inputs*) *ls*›
and *activation_tab_valid*: ‹*valid_activation_tab*$_l$ *atab*›
shows ‹
foldl (*predict*$_{layer\_l}$ (|*layers* = *ls*, *activation_tab* = *atab*|)) (*Some inputs*) *ls* =
Some (*foldl* (*predict*$_{layer_l}$_*impl* (|*layers* = *ls*, *activation_tab* = *atab*|)) *inputs ls*)
›
⟨*proof*⟩

lemma *predict*$_{seq_layer_l}$_*code* [*code*]:
assumes ‹*in_deg_NN N* = *length inputs*›
shows ‹*predict*$_{seq\_layer\_l}$ *N inputs* = *Some* (*predict*$_{seq_layer_l}$_*impl N inputs*)›
⟨*proof*⟩

lemma *predict*$_{seq_layer_l}$_*code*′ [*code*]:
assumes ‹*in_deg_NN N* = *length inputs*›
shows ‹*the* (*predict*$_{seq\_layer\_l}$ *N inputs*) = *predict*$_{seq_layer_l}$_*impl N inputs*›
⟨*proof*⟩

end

⟨*ML*⟩
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 ‹*valid_activation_tab*$_m$ *tab* = (∀ *f* ∈ *ran tab*. ∀ *xs*. *dim_vec xs* = *dim_vec* (*f xs*))›

lemma *valid_activation_preserves_dim*:
assumes ‹*valid_activation_tab*$_m$ *t*›
assumes ‹*t n* = *Some f*›
shows ‹*dim_vec xs* = *dim_vec* (*f xs*)›
⟨*proof*⟩

fun *layer_consistent*$_m$ :: (′*a vec*, ′*b*, ′*c mat*) *neural_network_seq_layers* ⇒ *nat* ⇒ (′*a vec*, ′*b*, ′*c mat*) *layer* ⇒ *bool*
where
‹*layer_consistent*$_m$ _ *nc* (*In l*) = (*o* < *units l* ∧ *nc* = *units l*)›
| ‹*layer_consistent*$_m$ _ *nc* (*Out l*) = (*o* < *units l* ∧ *nc* = *units l*)›
| ‹*layer_consistent*$_m$ *N nc* (*Activation l*) = ((*o* < *units l* ∧ *nc* = *units l*)
∧ (((*activation_tab N*) (*φ l*)) ≠ *None*))›

| ‹$layer\_consistent_m$ 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 $layers\_consistent_m$ **where**
‹$layers\_consistent_m$ N _ [] = True›
|‹$layers\_consistent_m$ N w (l#ls) = (($layer\_consistent_m$ N w l) ∧ ($layers\_consistent_m$ N (out_deg_layer l) ls))›

lemma $layer\_consistent_m\_activation\_tab\_const$:
‹$layer\_consistent_m$ N nc l = $layer\_consistent_m$ (|layers = ls, activation_tab = activation_tab N|) nc l›
⟨proof⟩

lemma $layers\_consistent_m\_activation\_tab\_const$:
‹$layers\_consistent_m$ N nc ls = $layers\_consistent_m$ (|layers = ls', activation_tab = activation_tab N|) nc ls›
⟨proof⟩

lemma $layers\_consistent_mAll$:
assumes ‹$layers\_consistent_m$ N inputs (layers N)›
shows ‹∀ l ∈ set (layers N). ∃ n . $layer\_consistent_m$ N n l›
⟨proof⟩

lemma $layers\_consistent_mAll'$:
assumes ‹$layers\_consistent_m$ N (in_deg_NN N) (layers N)›
shows ‹∀ l ∈ set (layers N). ∃ n . $layer\_consistent_m$ N n l›
⟨proof⟩

locale $neural\_network\_sequential\_layers_m$ =
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: ‹$valid\_activation\_tab_m$ (activation_tab N)›
and layer_valid: ‹$layers\_consistent_m$ 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 $predict_{layer\_m}$::‹('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**
‹$predict_{layer\_m}$ N (Some vs) (In l) = (if $layer\_consistent_m$ N (dim_vec vs) (In l) then Some vs else None) ›
| ‹$predict_{layer\_m}$ N (Some vs) (Out l) = (if $layer\_consistent_m$ N (dim_vec vs) (Out l) then Some vs else None) ›
| ‹$predict_{layer\_m}$ N (Some vs) (Dense pl) = (if $layer\_consistent_m$ N (dim_vec vs) (Dense pl) then

(case activation_tab N (φ pl) of
None ⇒ None
| Some f ⇒ Some (f ((vs $_v*$ ω pl) + β pl))
) else None)›
| ‹predict$_{layer_m}$ N (Some vs) (Activation pl) = (if layer_consistent$_m$ N (dim_vec vs) (Activation pl) then
(case activation_tab N (φ pl) of
None ⇒ None
| Some f ⇒ Some (f vs)
) else None)›
| ‹predict$_{layer_m}$ _ None _ = None ›

fun
predict$_{layer_m}$_impl ::‹('a::{comm_ring} Matrix.vec, 'b, 'a Matrix.mat) neural_network_seq_layers ⇒ 'a Matrix.vec
⇒ ('a Matrix.vec, 'b, 'a Matrix.mat) layer ⇒ 'a Matrix.vec›
where
‹predict$_{layer_m}$_impl N vs (In l) = vs›
| ‹predict$_{layer_m}$_impl N vs (Out l) = vs›
| ‹predict$_{layer_m}$_impl N vs (Dense pl) = ((the (activation_tab N (φ pl))) ((vs $_v*$ ω pl) + β pl))›
| ‹predict$_{layer_m}$_impl N vs (Activation pl) = (the (activation_tab N (φ pl)) vs)›

lemma predict_layer_Some:
assumes ‹(layer_consistent$_m$ N (dim_vec xs) l)›
shows ‹(predict$_{layer_m}$ N (Some xs) l ≠ None) ›
⟨proof⟩

definition ‹predict$_{seq_layer_m}$ N inputs = foldl (predict$_{layer_m}$ N) (Some inputs) (layers N)›
definition ‹predict$_{seq_layer_m}$_impl N inputs = foldl (predict$_{layer_m}$_impl N) inputs (layers N)›

definition ‹predict$_{seq_layer_m}$' N inputs = map_option list_of_vec (predict$_{seq_layer_m}$ N (vec_of_list inputs))›

lemma predict$_{layer_l}$_impl_activation_tab_const: ‹predict$_{layer_m}$_impl N = predict$_{layer_m}$_impl (|layers = l, activation_tab = activation_tab N|)›
⟨proof⟩

lemma layers_consistent$_m$_layersN_const:
‹layers_consistent$_m$ N = layers_consistent$_m$ (|layers = ls', activation_tab = activation_tab N|)›
⟨proof⟩

lemma predit$_{layer_m}$_impl_eq:
assumes ‹layer_consistent$_m$ N (dim_vec inputs) l›
shows ‹predict$_{layer_m}$ N (Some inputs) l = Some (predict$_{layer_m}$_impl N inputs l)›
⟨proof⟩

lemma valid_activation_preserves_length:
assumes ‹valid_activation_tab$_m$ t›
assumes ‹t n = Some f›
shows ‹dim_vec xs = dim_vec (f xs)›
⟨proof⟩

lemma fold_predict_m_strict: ‹(foldl (predict$_{layer_m}$ N) None ls) = None›

⟨*proof*⟩

lemmas [*nn_layer*] = *predict*$_{layer\_m}$.*simps predict_layer_Some fold_predict_m_strict*

lemma *predict*$_{layer\_m}$*_activation_tab*: **assumes** *activation_tab N* = *activation_tab N′* **shows**
‹*predict*$_{layer\_m}$ *N x xs* = *predict*$_{layer\_m}$ *N′ x xs*›
⟨*proof*⟩

lemma *predict*$_{layer\_m}$*_activation_tab_const*: ‹*predict*$_{layer\_m}$ *N* = *predict*$_{layer\_m}$ (|*layers* = *l*, *activation_tab* = *activation_tab N*|)›
⟨*proof*⟩

context *neural_network_sequential_layers*$_m$ **begin**
lemma *img_None_1*: **assumes** ‹((*predict*$_{seq\_layer\_m}$ *N xs*) ≠ *None*)› **shows** ‹(*dim_vec xs* = (*in_deg_NN N*))›
⟨*proof*⟩

lemma *img_None_2′*:
assumes *a0*: ‹*layers′* ≠ [] ›
and *a4*: ‹*valid_activation_tab*$_m$ *activation_tab′*›
and *a1*: ‹*layers_consistent*$_m$ (|*layers* = [], *activation_tab*=*activation_tab′* |) (*dim_vec xs*) *layers′*›
shows ‹*foldl* (*predict*$_{layer\_m}$ (|*layers* = [], *activation_tab* = *activation_tab′*|)) (*Some xs*) *layers′* ≠ *None*›
⟨*proof*⟩

lemma *img_None_2*:
assumes ‹*dim_vec xs* = *in_deg_NN N*›
shows ‹((*predict*$_{seq\_layer\_m}$ *N xs*) ≠ *None*)›
⟨*proof*⟩

lemma *img_None*: ‹((*predict*$_{seq\_layer\_m}$ *N xs*) ≠ *None*) = (*dim_vec xs* = *in_deg_NN N*)›
⟨*proof*⟩

lemma *img_Some*: ‹(∃ *y*. (*predict*$_{seq\_layer\_m}$ *N xs*) = *Some y*) = (*dim_vec xs* = *in_deg_NN N*)›
⟨*proof*⟩

lemma *img_deg*: ‹(∃ *y*. ((*predict*$_{seq\_layer\_m}$ *N xs*) = *Some y*) ⟶ (*dim_vec y* = *out_deg_NN N*))›
⟨*proof*⟩

lemma *aux_length*: *o* < *units x3* ⟹
o < *dim_vec inputs* ⟹
dim_vec (*β x3*) = *units x3* ⟹
dim_col (*ω x3*) = *units x3* ⟹
dim_row (*ω x3*) = *dim_vec inputs* ⟹
layers_consistent$_m$ (|*layers* = [], *activation_tab* = *atab*|) (*units x3*) *xs* ⟹
atab (*φ x3*) = *Some y* ⟹ *valid_activation_tab*$_m$ *atab* ⟹ *layers_consistent*$_m$ (|*layers* = [], *activation_tab* = *atab*|)
(*dim_vec* (*y* (*inputs* $_v$∗ *ω x3* + *β x3*))) *xs*
⟨*proof*⟩

lemma *pred_mat_impl_aux′*:
assumes ‹*ls* ≠ []›
and *layer_valid*: ‹*layers_consistent*$_m$ (|*layers* = [], *activation_tab* = *atab*|) (*dim_vec inputs*) *ls*›
and *activation_tab_valid*: ‹*valid_activation_tab*$_m$ *atab*›
shows ‹
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* = *name l*, *units* = *units l*, *φ* = *φ l*|)›
| ‹*layer_list_to_matrix* (*Dense l*) = (*let dimc* = *length* (*List.hd* (*ω l*)) *in*
(*Dense* (|*name* = *name l*, *units* = *units l*, *φ* = *φ l*,

$\beta = vec\_of\_list\ (\beta\ l), \omega = transpose\_mat\ (mat\_of\_rows\_list\ dimc\ (\omega\ l))\ \|)\!)\!)$›

fun
layer_matrix_to_list::‹('*a Matrix.vec*, '*b*, '*a Matrix.mat*) *layer* ⇒ ('*a list*, '*b*, '*a list list*) *layer*›
where
‹*layer_matrix_to_list* (*In l*) = (*In l*)›
| ‹*layer_matrix_to_list* (*Out l*) = (*Out l*)›
| ‹*layer_matrix_to_list* (*Activation l*) = (*Activation* (|*name* = *name l*, *units* = *units l*, φ = φ *l*|)›
| ‹*layer_matrix_to_list* (*Dense l*) = (*Dense* (|*name* = *name l*, *units* = *units l*, φ = φ *l*,
$\beta = list\_of\_vec\ (\beta\ l), \omega = mat\_to\_list\ (transpose\_mat\ \ (\omega\ l))\ \|)\!)$›

definition *activation_list_to_matrix*:: ‹('*b* ⇒ (('*a list* ⇒ '*a list*) *option*)) ⇒ ('*b* ⇒ (('*a Matrix.vec* ⇒ '*a Matrix.vec*) *option*))›
where
activation_list_to_matrix a = *map_option* (λf . *vec_of_list* ∘ *f* ∘ *list_of_vec*) ∘ *a*

definition *activation_matrix_to_list*:: ‹('*b* ⇒ (('*a Matrix.vec* ⇒ '*a Matrix.vec*) *option*)) ⇒ ('*b* ⇒ (('*a list* ⇒ '*a list*) *option*))›
where
activation_matrix_to_list a = *map_option* (λf . *list_of_vec* ∘ *f* ∘ *vec_of_list*) ∘ *a*

definition
nn_list_to_matrix::('*a list*, '*b*, '*a list list*) *neural_network_seq_layers* ⇒ ('*a Matrix.vec*, '*b*, '*a mat*) *neural_network_seq_layers*
where
‹*nn_list_to_matrix N* = (|*layers* = *map layer_list_to_matrix* (*layers N*),
activation_tab = *activation_list_to_matrix* (*activation_tab N*)|)›

definition
nn_matrix_to_list::('*a Matrix.vec*, '*b*, '*a mat*) *neural_network_seq_layers* ⇒ ('*a list*, '*b*, '*a list list*) *neural_network_seq_layers*
where
‹*nn_matrix_to_list N* = (|*layers* = *map layer_matrix_to_list* (*layers N*),
activation_tab = *activation_matrix_to_list* (*activation_tab N*)|)›

5.3.2 Converting Between List/Matrix-based Representations Preserves Consistency

lemma *layer_list_matrix_inverse*:
‹*layer_consistent*$_l$ *N n l* ⟹ *layer_matrix_to_list* (*layer_list_to_matrix l*) = *l*›
⟨*proof*⟩

lemma *layer_list_list_inverse*:
‹*layer_consistent*$_m$ *N n l* ⟹ *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*: ‹*activation_list_to_matrix* (*activation_matrix_to_list a*) *x* = *a x*›
⟨*proof*⟩

lemma *activation_matrix_inverse'*: ‹*activation_list_to_matrix* (*activation_matrix_to_list a*) = *a*›
⟨*proof*⟩

lemma *is_In_seq_l_eq_m*:
assumes ‹(*layers N*) ≠ []›
shows ‹*isIn* (*List.hd* (*layers N*)) = *isIn* (*List.hd* (*layers* (*nn_list_to_matrix N*)))›
⟨*proof*⟩

lemma *is_Out_seq_l_eq_m*:
assumes ‹(*layers N*) ≠ []›
shows ‹*isOut* (*last* (*layers N*)) = *isOut* (*last* (*layers* (*nn_list_to_matrix N*)))›
⟨*proof*⟩

lemma *is_Internal_seq_l_eq_m*:
assumes ‹(*layers N*) ≠ []›
shows ‹*list_all isInternal* ((*List.tl o butlast*) (*layers N*)) = *list_all isInternal* ((*List.tl o butlast*) (*layers* (*nn_list_to_matrix N*)))›
⟨*proof*⟩

lemma *valid_activation_tab_seq_l_imp_m*:
‹*valid_activation_tab*$_l$ (*activation_tab N*) ⟹ *valid_activation_tab*$_m$ (*activation_tab* (*nn_list_to_matrix N*))›
⟨*proof*⟩

lemma *layers_consistent_seq_l_imp_m*:
assumes ‹*layers_consistent*$_l$ *N n* (*layers N*)›
shows ‹*layers_consistent*$_m$ (*nn_list_to_matrix N*) *n* (*layers* (*nn_list_to_matrix N*)) ›
⟨*proof*⟩

lemma *in_deg_seq_l_eq_m*: ‹*in_deg_NN N* = (*in_deg_NN* (*nn_list_to_matrix N*))›
⟨*proof*⟩

lemma *is_In_seq_m_eq_l*:
assumes ‹(*layers N*) ≠ []›
shows ‹*isIn* (*List.hd* (*layers N*)) = *isIn* (*List.hd* (*layers* (*nn_matrix_to_list N*)))›
⟨*proof*⟩

lemma *is_Out_seq_m_eq_l*:
assumes ‹(*layers N*) ≠ []›
shows ‹*isOut* (*last* (*layers N*)) = *isOut* (*last* (*layers* (*nn_matrix_to_list N*)))›
⟨*proof*⟩

lemma *is_Internal_seq_m_eq_l*:
assumes ‹(*layers N*) ≠ []›
shows ‹*list_all isInternal* ((*List.tl o butlast*) (*layers N*)) = *list_all isInternal* ((*List.tl o butlast*) (*layers* (*nn_matrix_to_list N*)))›
⟨*proof*⟩

lemma *valid_activation_tab_seq_m_imp_l*:
‹*valid_activation_tab*$_m$ (*activation_tab N*) ⟹ *valid_activation_tab*$_l$ (*activation_tab* (*nn_matrix_to_list N*))›
⟨*proof*⟩

lemma *layers_consistent_seq_m_imp_l*:
assumes ‹*layers_consistent*$_m$ *N n* (*layers N*)›
shows ‹ *layers_consistent*$_l$ (*nn_matrix_to_list N*) *n* (*layers* (*nn_matrix_to_list N*)) ›
⟨*proof*⟩

lemma *in_deg_seq_m_eq_l*: ‹*in_deg_NN N* = (*in_deg_NN* (*nn_matrix_to_list N*))›
⟨*proof*⟩

theorem *neural_network_sequential_l_m*:
‹*neural_network_sequential_layers*$_l$ *N* ⟹ *neural_network_sequential_layers*$_m$ (*nn_list_to_matrix N*)›
⟨*proof*⟩

theorem *neural_network_sequential_m_l*:
‹*neural_network_sequential_layers*$_m$ *N* ⟹ *neural_network_sequential_layers*$_l$ (*nn_matrix_to_list N*)›
⟨*proof*⟩

lemma *matrix_list_inverse*:
assumes ‹*layers_consistent*$_l$ *N n* (*layers N*)›
shows ‹*nn_matrix_to_list* (*nn_list_to_matrix N*) = *N*›
⟨*proof*⟩

lemma *list_matrix_inverse*:
assumes ‹*layers_consistent*$_m$ *N n* (*layers N*)›
shows ‹*nn_list_to_matrix* (*nn_matrix_to_list N*) = *N*›
⟨*proof*⟩

lemma *square_nth_nth_id*:‹
∀ *w* ∈ *set ws*. *length w* = *length ws* ⟹
(*map* (λ*i*. (*map* (λ*ia*. *ws* ! *i* ! *ia*) [*o*..<*length ws*])) [*o*..<*length ws*]) = *ws*
›
⟨*proof*⟩

lemma *nth_map_f*: ‹*map* ((λ *i*. *f* (*xs* ! *i*))) [*o*..<*length xs*] = *map f xs*›
⟨*proof*⟩

lemma *square_nth_nth_id_f*:‹
∀ *w* ∈ *set ws*. *length w* = *length ws* ⟹
(*map* (λ*i*. (*map* (λ*ia*. *f* (*ws* ! *i* ! *ia*)) [*o*..<*length ws*])) [*o*..<*length ws*]) = *map* (*map f*) *ws*›
⟨*proof*⟩

lemma *F*:‹ *length* (*ws*::'*a*::{*comm_ring*} *list*) = *length Inputs* ⟹ *map* (λ*ia*. *ws* ! *ia* ∗ *Inputs* ! *ia*) [*o*..<*length Inputs*] = *map2* (∗) *Inputs ws*›
⟨*proof*⟩

lemma *list_singleton*: ‹*length xs* = *1* ⟹ ∃ *e*. *xs* = [*e*] ›
⟨*proof*⟩

lemma *activation_list_to_matrix_eq*:
‹*predict*$_{layer\_l}$ *N* (*Some* (*vs*::'*a*::*comm_ring list*)) (*Activation pl*) =
map_option list_of_vec (*predict*$_{layer\_m}$ (*nn_list_to_matrix N*) (*Some* (*vec_of_list vs*)) ((*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 ‹*dim_row M* $\neq$ *o*›
shows ‹*mat_to_list M* $\neq$ []›
⟨*proof*⟩

lemma *map2_to_map_idx_eq*: ‹*length xs* = *length ys* $\Longrightarrow$ (*map2* (∗) *xs* (*ys*)) = *map* (λ *i*. *xs*!*i* ∗ *ys*!*i*) [*o*..< *length xs*]›
⟨*proof*⟩

lemma *map2_to_map_idx*: ‹(*map2* (∗) *xs* (*ys*)) = *map* (λ *i*. *xs*!*i* ∗ *ys*!*i*) [*o*..< *min* (*length xs*) (*length ys*)]›
⟨*proof*⟩

lemma *length_list_transpose_mat*: ‹*o* < *dim_row M* $\Longrightarrow$ *o* < *dim_col M* $\Longrightarrow$ *length* (*List.transpose* (*mat_to_list M*)) = *dim_col M*›
⟨*proof*⟩

lemma *map_sum_list_idx*: ‹
map (λ*m*. *sum_list* (*map2* (∗) *m* (*list_of_vec v*))) (*List.transpose* (*mat_to_list M*))
= *map* (λ*i*. *sum_list* (*map2* (∗) ((*List.transpose* (*mat_to_list M*))!*i*) (*list_of_vec v*))) [*o*..<*length* (*List.transpose* (*mat_to_list M*))]›
⟨*proof*⟩

lemma *vec_mult_mat_list*:
assumes ‹∀ *as*∈*set* (*mat_to_list M*). *length as* = *dim_col M*›
and ‹*dim_vec v* = *dim_row M*›
and ‹*dim_col M* $\neq$ *o*›
and ‹*dim_row M* $\neq$ *o*›
shows ‹ (*v*::'*a*::*comm_ring vec*) $_v*$ *M* = *vec_of_list* (*map* (λ*m*. *sum_list* (*map2* (∗) *m* (*list_of_vec v*))) (*mat_to_list* M^T))›
⟨*proof*⟩

lemma *hd_length_inputs*: ‹*o* < *units x3* $\Longrightarrow$
length (β *x3*) = *units x3* $\Longrightarrow$ *length* (ω *x3*) = *units x3* $\Longrightarrow$ ∀ *r*∈*set* (ω *x3*). *length r* = *length Inputs* $\Longrightarrow$ *length Inputs* = *length* (*List.hd* (ω *x3*))›
⟨*proof*⟩

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

lemma *In_l_to_m_eq*:
‹*predict*$_{layer\_l}$ *N* (*Some vs*) (*In l*) = *map_option list_of_vec* (*predict*$_{layer\_m}$ (*nn_list_to_matrix N*) (*Some* (*vec_of_list vs*)) (*layer_list_to_matrix* (*In l*)))›
⟨*proof*⟩

lemma *In_l_to_m_eq'*:
‹*predict*$_{layer\_m}$ (*nn_list_to_matrix N*) (*Some* (*vec_of_list vs*)) (*layer_list_to_matrix* (*In l*)) = *map_option vec_of_list* (*predict*$_{layer\_l}$ *N* (*Some vs*) (*In l*)) ›
⟨*proof*⟩

lemma *Out_l_to_m_eq*:
‹*predict*$_{layer\_l}$ *N* (*Some vs*) (*Out l*) = *map_option list_of_vec* (*predict*$_{layer\_m}$ (*nn_list_to_matrix N*) (*Some* (*vec_of_list vs*)) (*layer_list_to_matrix* (*Out l*)))›
⟨*proof*⟩

lemma *Out_l_to_m_eq'*:
‹*predict*$_{layer\_m}$ (*nn_list_to_matrix N*) (*Some* (*vec_of_list vs*)) (*layer_list_to_matrix* (*Out l*)) = *map_option vec_of_list* (*predict*$_{layer\_l}$ *N* (*Some vs*) (*Out l*))›

⟨*proof*⟩

lemma *Dense_l_to_m_eq*:
assumes ‹*layer_consistent*$_l$ *N* (*length vs*) (*Dense l*)›
shows ‹*predict*$_{layer\_l}$ *N* (*Some* (*vs*::'*a*::*comm_ring list*)) (*Dense l*)
= *map_option list_of_vec* (*predict*$_{layer\_m}$ (*nn_list_to_matrix N*) (*Some* (*vec_of_list vs*)) (*layer_list_to_matrix* (*Dense l*)))›
⟨*proof*⟩

lemma *Dense_l_to_m_eq'*:
assumes ‹*layer_consistent*$_l$ *N* (*length vs*) (*Dense l*)›
shows ‹*predict*$_{layer\_m}$ (*nn_list_to_matrix N*) (*Some* (*vec_of_list vs*)) (*layer_list_to_matrix* (*Dense l*))
= *map_option vec_of_list* (*predict*$_{layer\_l}$ *N* (*Some* (*vs*::'*a*::*comm_ring list*)) (*Dense l*))›
⟨*proof*⟩

lemma *Activation_l_to_m_eq*:
‹*predict*$_{layer\_l}$ *N* (*Some vs*) (*Activation l*)
= *map_option list_of_vec* (*predict*$_{layer\_m}$ (*nn_list_to_matrix N*) (*Some* (*vec_of_list vs*)) (*layer_list_to_matrix* (*Activation l*)))›
⟨*proof*⟩

lemma *Activation_l_to_m_eq'*:
‹*predict*$_{layer\_m}$ (*nn_list_to_matrix N*) (*Some* (*vec_of_list vs*)) (*layer_list_to_matrix* (*Activation l*))
= *map_option vec_of_list* (*predict*$_{layer\_l}$ *N* (*Some vs*) (*Activation l*))›
⟨*proof*⟩

lemma *aux1*: ‹
⋀*y*. *l* = *Dense x3* ⟹
(⋀*Inputs*.
layers_consistent$_l$ (|*layers* = *lo*, *activation_tab* = *activation_tab'*|) (*length Inputs*) *layers'* ⟹
foldl (*predict*$_{layer\_l}$ (|*layers* = *l1*, *activation_tab* = *activation_tab'*|)) (*Some Inputs*) *layers'* =
map_option list_of_vec (*foldl* (*predict*$_{layer\_m}$ (*nn_list_to_matrix* (|*layers* = *l2*, *activation_tab* = *activation_tab'*|)))
(*Some* (*vec_of_list Inputs*)) (*layers* (*nn_list_to_matrix* (|*layers* = *layers'*, *activation_tab* = *a2*|))))) ⟹
valid_activation_tab$_l$ *activation_tab'* ⟹
o < *units x3* ⟹
Inputs ≠ [] ⟹
length (*LayerRecord*.β *x3*) = *units x3* ⟹
length (*LayerRecord*.ω *x3*) = *units x3* ⟹
∀ *r*∈*set* (*LayerRecord*.ω *x3*). *length r* = *length Inputs* ⟹
layers_consistent$_l$ (|*layers* = *lo*, *activation_tab* = *activation_tab'*|) (*units x3*) *layers'* ⟹
activation_tab' (*ActivationRecord*.φ *x3*) = *Some y* ⟹
foldl (*predict*$_{layer\_l}$ (|*layers* = *l1*, *activation_tab* = *activation_tab'*|)) (*Some* (*y* (*map2* (+) (*map* ((λ*vs'*. ∑ (*x*, *y*)←*vs'*. *x* ∗ *y*) ∘ *zip Inputs*) (*LayerRecord*.ω *x3*)) (*LayerRecord*.β *x3*)))) *layers'* =
map_option list_of_vec
(*foldl* (*predict*$_{layer\_m}$ (*nn_list_to_matrix* (|*layers* = *l2*, *activation_tab* = *activation_tab'*|))) (*Some* (*vec_of_list* (*y*
(*map2* (+) (*map* ((λ*vs'*. ∑ (*x*, *y*)←*vs'*. *x* ∗ *y*) ∘ *zip Inputs*) (*LayerRecord*.ω *x3*)) (*LayerRecord*.β *x3*)))))
(*map layer_list_to_matrix layers'*)) ›
⟨*proof*⟩

lemma *precdict_seq_l_eq_m'*:

assumes ‹*layers_consistent*$_l$ (|*layers* = *l0*, *activation_tab* = *activation_tab'*|) (*length* (*Inputs*::'*a*::*comm_ring list*)) *layers'*›
and ‹*valid_activation_tab*$_l$ *activation_tab'*›
shows ‹*foldl* (*predict*$_{layer\_l}$ (|*layers* = *l1*, *activation_tab* = *activation_tab'*|)) (*Some* (*Inputs*)) (*layers* (|*layers* = *layers'*, *activation_tab* =*a1*|)) =
map_option list_of_vec
(*foldl* (*predict*$_{layer\_m}$ (*nn_list_to_matrix* (|*layers* = *l2*, *activation_tab* = *activation_tab'*|))) (*Some* (*vec_of_list Inputs*))
(*layers* (*nn_list_to_matrix* (|*layers* = *layers'*, *activation_tab* = *a2*|))))›
⟨*proof*⟩

theorem *precdict_seq_l_eq_m*:
assumes ‹*layers_consistent*$_l$ *N* (*length Inputs*) (*layers N*)›
and ‹*valid_activation_tab*$_l$ (*activation_tab N*)›
shows ‹*predict*$_{seq\_layer\_l}$ *N* (*Inputs*::'*a*::*comm_ring list*) = *predict*$_{seq\_layer\_m}$' (*nn_list_to_matrix N*) *Inputs*›
⟨*proof*⟩

lemma *In_m_to_l_eq*:
‹*predict*$_{layer\_m}$ *N* (*Some vs*) (*In l*) = *map_option vec_of_list* (*predict*$_{layer\_l}$ (*nn_matrix_to_list N*) (*Some* (*list_of_vec vs*)) (*layer_matrix_to_list* (*In l*)))›
⟨*proof*⟩

lemma *In_m_to_l_eq'*:
‹*predict*$_{layer\_l}$ (*nn_matrix_to_list N*) (*Some* (*list_of_vec vs*)) (*layer_matrix_to_list* (*In l*)) = *map_option list_of_vec* (*predict*$_{layer\_m}$ *N* (*Some vs*) (*In l*)) ›
⟨*proof*⟩

lemma *Out_m_to_l_eq*:
‹*predict*$_{layer\_m}$ *N* (*Some vs*) (*Out l*) = *map_option vec_of_list* (*predict*$_{layer\_l}$ (*nn_matrix_to_list N*) (*Some* (*list_of_vec vs*)) (*layer_matrix_to_list* (*Out l*)))›
⟨*proof*⟩

lemma *Out_m_to_l_eq'*:
‹*predict*$_{layer\_l}$ (*nn_matrix_to_list N*) (*Some* (*list_of_vec vs*)) (*layer_matrix_to_list* (*In l*)) = *map_option list_of_vec* (*predict*$_{layer\_m}$ *N* (*Some vs*) (*Out l*)) ›
⟨*proof*⟩

lemma *Dense_m_to_l_eq*:
assumes ‹*layer_consistent*$_m$ *N* (*dim_vec vs*) (*Dense l*)›
shows ‹*predict*$_{layer\_m}$ *N* (*Some* (*vs*::'*a*::*comm_ring Matrix.vec*)) (*Dense l*)
= *map_option vec_of_list* (*predict*$_{layer\_l}$ (*nn_matrix_to_list N*) (*Some* (*list_of_vec vs*)) (*layer_matrix_to_list* (*Dense l*)))›
⟨*proof*⟩

lemma *Dense_m_to_l_eq'*:
assumes ‹*layer_consistent*$_m$ *N* (*dim_vec vs*) (*Dense l*)›
shows ‹*predict*$_{layer\_l}$ (*nn_matrix_to_list N*) (*Some* (*list_of_vec vs*)) (*layer_matrix_to_list* (*Dense l*))
= *map_option list_of_vec* (*predict*$_{layer\_m}$ *N* (*Some* (*vs*::'*a*::*comm_ring Matrix.vec*)) (*Dense l*))›
⟨*proof*⟩

lemma *Activation_m_to_l_eq*:
‹*predict*$_{layer\_m}$ *N* (*Some vs*) (*Activation l*)
= *map_option vec_of_list* (*predict*$_{layer\_l}$ (*nn_matrix_to_list N*) (*Some* (*list_of_vec vs*)) (*layer_matrix_to_list* (*Activation*

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 *precdict_seq_m_eq_l'*:
assumes ‹*layers_consistent*$_m$ (|*layers* = *lo*, *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 *precdict_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 *precdict_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

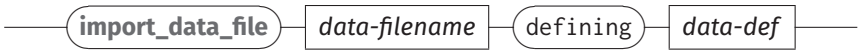
end

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:

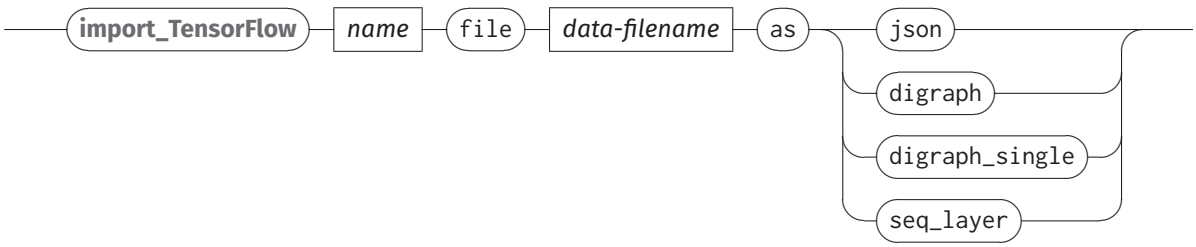


where *data-filename* is the path (file name) of the data file that should be read and *data-def* is the name the HOL definition representing the data is bound to. The data file should be a simple two-dimensional array of real numbers as, e.g., produced by NumPy's [11] `savetxt` command:

```python
import numpy as np
training_data = np.array([[0,0],[0,1],[1,0],[1,1]], "float32")
np.savetxt('training_data.out', training_data)
```

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:

```python
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** $m\_N0 \equiv In\ 0$
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 $m\_N9 \equiv Neuron\ (\lfloor \varphi = Identity, \alpha = 1, \beta = 7082077 \,/\, 1000000000, uid = 9 \rfloor)$
definition $m\_N10 \equiv Neuron\ (\lfloor \varphi = Identity, \alpha = 1, \beta = 107544795 \,/\, 1000000000, uid = 10 \rfloor)$
definition $m\_N11 \equiv Neuron\ (\lfloor \varphi = Identity, \alpha = 1, \beta = -\ 15743796 \,/\, 1000000000, uid = 11 \rfloor)$
definition $m\_N12 \equiv Neuron\ (\lfloor \varphi = Identity, \alpha = 1, \beta = 47920802 \,/\, 100000000, uid = 12 \rfloor)$
definition $m\_N13 \equiv Neuron\ (\lfloor \varphi = Identity, \alpha = 1, \beta = -\ 16364478 \,/\, 100000000, uid = 13 \rfloor)$
definition $m\_N14 \equiv Neuron\ (\lfloor \varphi = Identity, \alpha = 1, \beta = -\ 24132763 \,/\, 100000000, uid = 14 \rfloor)$
definition $m\_N15 \equiv Neuron\ (\lfloor \varphi = Identity, \alpha = 1, \beta = -\ 30579916 \,/\, 100000000, uid = 15 \rfloor)$
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_N0_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 $m\_Neurons = [m\_N0, m\_N1, m\_N2, m\_N3, m\_N4, m\_N5, m\_N6, m\_N7, m\_N8, m\_N9, m\_N10, m\_N11,$
$m\_N12, m\_N13, m\_N14, m\_N15, m\_N16, m\_N17, m\_N18, m\_N19]$

Definition: Edges **definition** $m\_E12\_16 \equiv (\lfloor \omega = 1, tl = m\_N12, hd = m\_N16 \rfloor)$
definition $m\_E13\_17 \equiv (\lfloor \omega = 1, tl = m\_N13, hd = m\_N17 \rfloor)$
definition $m\_E14\_18 \equiv (\lfloor \omega = 1, tl = m\_N14, hd = m\_N18 \rfloor)$
definition $m\_E15\_19 \equiv (\lfloor \omega = 1, tl = m\_N15, hd = m\_N19 \rfloor)$
definition $m\_E9\_12 \equiv (\lfloor \omega = 8217673 \,/\, 200000000, tl = m\_N9, hd = m\_N12 \rfloor)$

definition *m_E10_12* ≡ ⟨|ω = *2972081* / *20000000*, *tl* = *m_N10*, *hd* = *m_N12*|⟩
definition *m_E11_12* ≡ ⟨|ω = *2445593* / *10000000*, *tl* = *m_N11*, *hd* = *m_N12*|⟩
definition *m_E9_13* ≡ ⟨|ω = − (*11993983* / *5000000*), *tl* = *m_N9*, *hd* = *m_N13*|⟩
definition *m_E10_13* ≡ ⟨|ω = − (*3894687* / *50000000*), *tl* = *m_N10*, *hd* = *m_N13*|⟩
definition *m_E11_13* ≡ ⟨|ω = *646179* / *1250000*, *tl* = *m_N11*, *hd* = *m_N13*|⟩
definition *m_E9_14* ≡ ⟨|ω = − (*2323241* / *5000000*), *tl* = *m_N9*, *hd* = *m_N14*|⟩
definition *m_E10_14* ≡ ⟨|ω = *10928257* / *10000000*, *tl* = *m_N10*, *hd* = *m_N14*|⟩
definition *m_E11_14* ≡ ⟨|ω = − (*7042477* / *5000000*), *tl* = *m_N11*, *hd* = *m_N14*|⟩
definition *m_E9_15* ≡ ⟨|ω = *19465483* / *10000000*, *tl* = *m_N9*, *hd* = *m_N15*|⟩
definition *m_E10_15* ≡ ⟨|ω = − (*9524061* / *10000000*), *tl* = *m_N10*, *hd* = *m_N15*|⟩
definition *m_E11_15* ≡ ⟨|ω = − (*31743723* / *50000000*), *tl* = *m_N11*, *hd* = *m_N15*|⟩
definition *m_E0_9* ≡ ⟨|ω = *3342313* / *5000000*, *tl* = *m_N0*, *hd* = *m_N9*|⟩
definition *m_E1_9* ≡ ⟨|ω = − (*12952799* / *10000000*), *tl* = *m_N1*, *hd* = *m_N9*|⟩
definition *m_E2_9* ≡ ⟨|ω = − (*1428979* / *5000000*), *tl* = *m_N2*, *hd* = *m_N9*|⟩
definition *m_E3_9* ≡ ⟨|ω = *8650103* / *5000000*, *tl* = *m_N3*, *hd* = *m_N9*|⟩
definition *m_E4_9* ≡ ⟨|ω = *63918763* / *100000000*, *tl* = *m_N4*, *hd* = *m_N9*|⟩
definition *m_E5_9* ≡ ⟨|ω = − (*6959659* / *5000000*), *tl* = *m_N5*, *hd* = *m_N9*|⟩
definition *m_E6_9* ≡ ⟨|ω = − (*9054079* / *20000000*), *tl* = *m_N6*, *hd* = *m_N9*|⟩
definition *m_E7_9* ≡ ⟨|ω = *13654941* / *10000000*, *tl* = *m_N7*, *hd* = *m_N9*|⟩
definition *m_E8_9* ≡ ⟨|ω = − (*18450487* / *100000000*), *tl* = *m_N8*, *hd* = *m_N9*|⟩
definition *m_E0_10* ≡ ⟨|ω = *314303* / *5000000*, *tl* = *m_N0*, *hd* = *m_N10*|⟩
definition *m_E1_10* ≡ ⟨|ω = *915709* / *2500000*, *tl* = *m_N1*, *hd* = *m_N10*|⟩
definition *m_E2_10* ≡ ⟨|ω = *6922799* / *10000000*, *tl* = *m_N2*, *hd* = *m_N10*|⟩
definition *m_E3_10* ≡ ⟨|ω = − (*9399607* / *25000000*), *tl* = *m_N3*, *hd* = *m_N10*|⟩
definition *m_E4_10* ≡ ⟨|ω = *15055849* / *100000000*, *tl* = *m_N4*, *hd* = *m_N10*|⟩
definition *m_E5_10* ≡ ⟨|ω = *10981513* / *10000000*, *tl* = *m_N5*, *hd* = *m_N10*|⟩
definition *m_E6_10* ≡ ⟨|ω = *3420911* / *200000000*, *tl* = *m_N6*, *hd* = *m_N10*|⟩
definition *m_E7_10* ≡ ⟨|ω = *7420693* / *10000000*, *tl* = *m_N7*, *hd* = *m_N10*|⟩
definition *m_E8_10* ≡ ⟨|ω = *15639223* / *100000000*, *tl* = *m_N8*, *hd* = *m_N10*|⟩
definition *m_E0_11* ≡ ⟨|ω = *9863281* / *100000000*, *tl* = *m_N0*, *hd* = *m_N11*|⟩
definition *m_E1_11* ≡ ⟨|ω = *9530481* / *10000000*, *tl* = *m_N1*, *hd* = *m_N11*|⟩
definition *m_E2_11* ≡ ⟨|ω = *35006753* / *100000000*, *tl* = *m_N2*, *hd* = *m_N11*|⟩
definition *m_E3_11* ≡ ⟨|ω = *7897923* / *10000000*, *tl* = *m_N3*, *hd* = *m_N11*|⟩
definition *m_E4_11* ≡ ⟨|ω = − (*11627171* / *20000000*), *tl* = *m_N4*, *hd* = *m_N11*|⟩
definition *m_E5_11* ≡ ⟨|ω = *2839861* / *5000000*, *tl* = *m_N5*, *hd* = *m_N11*|⟩
definition *m_E6_11* ≡ ⟨|ω = *5311743* / *10000000*, *tl* = *m_N6*, *hd* = *m_N11*|⟩
definition *m_E7_11* ≡ ⟨|ω = − (*9090567* / *10000000*), *tl* = *m_N7*, *hd* = *m_N11*|⟩
definition *m_E8_11* ≡ ⟨|ω = − (*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
‹*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*]›

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* (*predict*$_{digraph\_single}$ *compass.NeuralNet* (*map_of_list* (*input*!0)) *E12_16*)) (*Some* (*predictions*!0!0))›
value ‹(*checkget_result_singleton 0.15* (*predict*$_{digraph\_single}$ *compass.NeuralNet* (*map_of_list* (*input*!0)) *E12_16*)) (*Some* (*predictions*!1!0))›
value ‹(*checkget_result_singleton 0.15* (*predict*$_{digraph\_single}$ *compass.NeuralNet* (*map_of_list* (*input*!0)) *E12_16*)) (*Some* (*predictions*!2!0))›
value ‹(*checkget_result_singleton 0.15* (*predict*$_{digraph\_single}$ *compass.NeuralNet* (*map_of_list* (*input*!0)) *E12_16*)) (*Some* (*predictions*!3!0))›

lemma *compass_truth_table_predict*:
‹(*predict*$_{digraph\_single}$ *compass.NeuralNet* (*map_of_list* (*input*!0)) *E12_16*) ≈[0.0001]≈$_s$ (*Some* (*predictions*!0!0))›
‹(*predict*$_{digraph\_single}$ *compass.NeuralNet* (*map_of_list* (*input*!1)) *E12_16*) ≈[0.0001]≈$_s$ (*Some* (*predictions*!1!0))›
‹(*predict*$_{digraph\_single}$ *compass.NeuralNet* (*map_of_list* (*input*!2)) *E12_16*) ≈[0.0001]≈$_s$ (*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* = *predict*$_{digraph\_single}$ *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* :: ‹*activation*$_{multi}$ ⇒ (*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 / 100000000000000000, − 959978699684143 / 10000000000000000, 7840137928724289 / 100000000000000000],
ω = [[− 28655481338500977 / 100000000000000000, 1398887038230896 / 10000000000000000, − 4396021068096161 / 10000000000000000,
− 3206970691680908 / 100000000000000000, − 25562143325805664 / 100000000000000000, 11852015256881714 / 10000000000000000,
6039865016937256 / 100000000000000000, − 16825008392333984 / 10000000000000000, − 413370318710804 / 100000000000000000],
[24456006288528442 / 100000000000000000, − 11522198915481567 / 100000000000000000, 4993317425251007 / 100000000000000000,
− 17345187664031982 / 100000000000000000, 48335906863212585 / 100000000000000000, 1511125922203064 / 10000000000000000,
− 36204618215560913 / 100000000000000000, 9508050084114075 / 100000000000000000, − 3617756962776184 / 100000000000000000],
[704086497426033 / 100000000000000000, − 51195383071899414 / 100000000000000000, − 34204763174057007 / 100000000000000000,
− 72454833984375 / 100000000000000, − 33541640639305115 / 100000000000000000, 12738076448440552 / 100000000000000000,
7601173520088196 / 100000000000000000, − 2638514041900635 / 100000000000000000, − 5478811264038086 / 1000000000000000000]]⦈)
definition *m_dense_2* =
Dense
(⦇*name* = *STR ''dense_2''*, *units* = 4, *φ* = *mIdentity*,
β = [39810407906770706 / 100000000000000000, 874686986207962 / 100000000000000000, − 4944610595703125 / 100000000000000000,
− 5116363242268562 / 100000000000000000],
ω = [[(9063153862953186 / 1000000000000000000::*real*), − 142851984500885 / 100000000000000, − 10823805332183838 / 100000000000000000],
[17654908895492554 / 100000000000000000, 1934271901845932 / 100000000000000000, 1214023232460022 / 1000000000000000000],

[− 17099318504333496 / 10000000000000000, − 7595149427652359 / 100000000000000000, − 12841564416885376 / 10000000000000000],
[− 615866482257843 / 1000000000000000, 1532884955406189 / 1000000000000000, 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*: ‹*ran m_φ_compass* = {*map identity*}›
⟨*proof*⟩

interpretation nn_{nor}: *neural_network_sequential_layers*$_l$ *m_NeuralNet*
⟨*proof*⟩

TensorFlow Import

declare[[*nn_proof_mode* = *eval*]]
import_TensorFlow *compass* **file** *model*/*model.json* **as** *seq_layer_list*
declare[[*nn_proof_mode* = *nbe*]]

thm *compass.Layers.dense_input_def*
thm *compass.Layers.dense_def*
thm *compass.Layers.OUTPUT_def*
thm *compass.layer_defs*
thm *compass.Layers_def*
thm *compass.φ_compass.simps*
thm *compass.NeuralNet_def*

thm *compass.neural_network_sequential_layers*$_l$*_axioms*

import_data_file *model*/*input.txt* **defining** *input*
import_data_file *model*/*predictions.txt* **defining** *predictions*

thm *input_def*
thm *predictions_def*

lemmas *digits_defs* = *compass.Layers_def*
lemmas *activation_defs* = *identity_def*

value *predict*$_{seq\_layer\_l}$ *NeuralNet* (*input*!0)

value ‹*checkget_result_list* 0.001 (*predict*$_{seq\_layer\_l}$ *NeuralNet* (*input*!0)) (*Some* (*predictions*!0))›
value ‹*checkget_result_list* 0.001 (*predict*$_{seq\_layer\_l}$ *NeuralNet* (*input*!1)) (*Some* (*predictions*!1))›
value ‹*checkget_result_list* 0.001 (*predict*$_{seq\_layer\_l}$ *NeuralNet* (*input*!2)) (*Some* (*predictions*!2))›
value ‹*checkget_result_list* 0.001 (*predict*$_{seq\_layer\_l}$ *NeuralNet* (*input*!3)) (*Some* (*predictions*!3))›

We convince ourselves that our Isabelle representation complies with the TensorFlow network by generating the same prediction, within 0.001 (accounted for as Isabelle uses perfect mathematical reals whereas TensorFlow uses 32-bit floating point numbers)

lemma *compass_predictions*:
‹(*predict*$_{seq\_layer\_l}$ *NeuralNet* (*input*!0)) ≈[0.001]≈$_l$ (*Some* (*predictions*!0))›
‹(*predict*$_{seq\_layer\_l}$ *NeuralNet* (*input*!1)) ≈[0.001]≈$_l$ (*Some* (*predictions*!1))›
‹(*predict*$_{seq\_layer\_l}$ *NeuralNet* (*input*!2)) ≈[0.001]≈$_l$ (*Some* (*predictions*!2))›
‹(*predict*$_{seq\_layer\_l}$ *NeuralNet* (*input*!3)) ≈[0.001]≈$_l$ (*Some* (*predictions*!3))›
⟨*proof*⟩

lemma ‹0.000001 ⊨$_l$ {*input*} (*predict*$_{seq\_layer\_l}$ *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*

definition *classify_as* :: ‹*real list* ⇒ *nat* ⇒ *bool*› **where**
‹*classify_as xs n* = (*Option.bind* (*predict*$_{seq\_layer\_l}$ *compass.NeuralNet xs*) *Prediction_Utils.pos_of_max* = *Some n*)›

lemma *c0*[*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]
⟨*proof*⟩

lemma *c3*[*simp*]: *compass*!3 = [1,0,1,
1,1,0,
1,1,1]
⟨*proof*⟩

lemma *classify_NW*: ‹*classify_as* (*compass*!0) 0›
⟨*proof*⟩
lemma *classify_NE*: ‹*classify_as* (*compass*!1) 1›
⟨*proof*⟩

lemma *classify_SE*: ‹*classify_as* (*compass*!2) 2›

⟨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 mIdentity* = *Some* (*map_vec identity*)›
| ‹*m_φ_compass* _ = *None*›

Definition: Layers **definition** *m_dense_input* = *In* (|*name* = *STR* ''*dense_input*'', *units* = 9|)
definition *m_dense* =
Dense
(|*name* = *STR* ''*dense*'', *units* = 3, *φ* = *mIdentity*,
β = *vec_of_list* [*9153944253921509* / *100000000000000000*, − *959978699684143* / *10000000000000000*, *7840137928724289* / *100000000000000000*],
ω = *mat_of_cols_list* 9
[[− *28655481338500977* / *100000000000000000*, *1398887038230896* / *10000000000000000*, − *4396021068096161* / *10000000000000000*,
− *3206970691680908* / *100000000000000000*, − *25562143325805664* / *100000000000000000*, *11852015256881714* / *10000000000000000*,
6039865016937256 / *10000000000000000*, − *16825008392333984* / *100000000000000000*, − *413370318710804* / *100000000000000000*],
[*24456006288528442* / *100000000000000000*, − *11522198915481567* / *10000000000000000*, *4993317425251007* / *10000000000000000*,
− *17345187664031982* / *100000000000000000*, *48335906863212585* / *100000000000000000*, *1511125922203064* / *10000000000000000*,
− *36204618215560913* / *100000000000000000*, *9508050084114075* / *100000000000000000*, − *3617756962776184* / *100000000000000000*],
[*704086497426033* / *100000000000000000*, − *51195383071899414* / *100000000000000000*, − *34204763174057007* / *100000000000000000*,

$-$ *72454833984375* / *100000000000000*, $-$ *33541640639305115* / *100000000000000000*, *12738076448440552* / *10000000000000000*,
7601173520088196 / *10000000000000000*, $-$ *2638514041900635* / *10000000000000000*, $-$ *5478811264038086* / *10000000000000000*]]⟫
definition *m_dense_2* =
Dense
(⦇*name* = *STR ''dense_2''*, *units* = 4, φ = *mIdentity*,
β = *vec_of_list* [*39810407906770706* / *1000000000000000000*, *874686986207962* / *10000000000000000*, $-$ *4944610595703125* / *100000000000000000*,
$-$ *5116363242268562* / *100000000000000000*],
ω = *mat_of_cols_list 3*
[[(*9063153862953186* / *10000000000000000*::*real*), $-$ *142851984500885* / *100000000000000*, $-$ *10823805332183838* / *10000000000000000*],
[*17654908895492554* / *10000000000000000*, *1934271901845932* / *10000000000000000*, *1214023232460022* / *1000000000000000*],
[$-$ *17099318504333496* / *10000000000000000*, $-$ *7595149427652359* / *100000000000000000*, $-$ *12841564416885376* / *10000000000000000*],
[$-$ *615866482257843* / *1000000000000000*, *1532884955406189* / *1000000000000000*, *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*: ‹*ran m_*φ*_compass* = {*map_vec identity*}›
⟨*proof*⟩

interpretation *nn*$_{nor}$: *neural_network_sequential_layers*$_m$ *m_NeuralNet*
⟨*proof*⟩

TensorFlow Import

declare[[*nn_proof_mode* = *eval*]]
import_TensorFlow *compass* **file** *model/model.json* **as** *seq_layer_matrix*
declare[[*nn_proof_mode* = *nbe*]]

find_theorems *name*:*compass*. *name*:φ *name*:*ran*

thm *compass.Layers.dense_input_def*
thm *compass.Layers.dense_def*
thm *compass.Layers.OUTPUT_def*
thm *compass.layer_defs*
thm *compass.Layers_def*
thm *compass.*φ*_compass.simps*
thm *compass.NeuralNet_def*
thm *compass.neural_network_sequential_layers*$_m$*_axioms*

import_data_file *model/input.txt* **defining** *input*
import_data_file *model/predictions.txt* **defining** *predictions*

thm *input_def*
thm *predictions_def*

value ‹(*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]≈$_l$ (*Some* (*predictions*!0))›
‹(*map_option list_of_vec* (*predict*$_{seq\_layer\_m}$ *NeuralNet* ((*vec_of_list* (*input*!1))))) ≈[0.001]≈$_l$ (*Some* (*predictions*!1))›
‹(*map_option list_of_vec* (*predict*$_{seq\_layer\_m}$ *NeuralNet* ((*vec_of_list* (*input*!2))))) ≈[0.001]≈$_l$ (*Some* (*predictions*!2))›
‹(*map_option list_of_vec* (*predict*$_{seq\_layer\_m}$ *NeuralNet* ((*vec_of_list* (*input*!3))))) ≈[0.001]≈$_l$ (*Some* (*predictions*!3))›
⟨*proof*⟩

lemma ‹0.000001 ⊨$_l$ {*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 *c0*[*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]
⟨proof⟩

lemma c3[simp]: compass!3 = [1,0,1,
                1,1,0,
                1,1,1]
⟨proof⟩


lemma compass_img_defined: ‹((predict_seq_layer_m compass.NeuralNet xs) ≠ None) = (length (list_of_vec xs) = 9)›
⟨proof⟩

definition classify_as :: ‹real Matrix.vec ⇒ nat ⇒ bool› where
‹classify_as xs n = (Option.bind (predict_seq_layer_m compass.NeuralNet xs) pos_of_max = Some n)›

lemma classify_NW: ‹classify_as (vec_of_list(compass!0)) 0›
⟨proof⟩
lemma classify_NE: ‹classify_as (vec_of_list(compass!1)) 1›
⟨proof⟩
lemma classify_SE: ‹classify_as (vec_of_list(compass!2)) 2›
⟨proof⟩
lemma classify_SW: ‹classify_as (vec_of_list(compass!3)) 3›
⟨proof⟩

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).

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

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

$$y_k = \varphi\left(\beta \sum_{j=0}^{m} w_{k_j} x_{k_j}\right) \tag{8.1}$$

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].

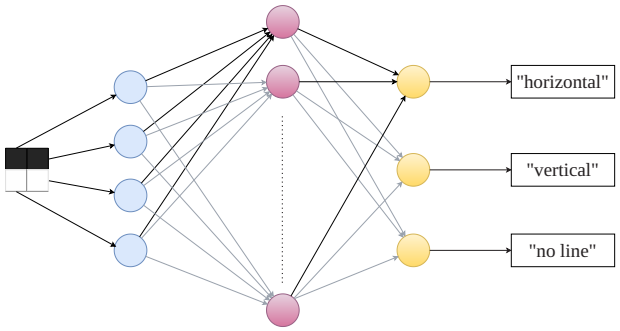


Figure 8.2: Neural network for classifying lines in 2×2 pixel grey scale images.

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*]:
```
  ‹predict_seq_layer_l grid.NeuralNet xs = (case xs of
  [x3, x2, x1, x0] ⇒ let y = 2 * (x3 + (x2 * 2 + (x1 * 4 + x0 * 8)))
   in Some (
    [(if y − 7 ≤ 0 then 0 else 1) +
      ((if y − 11 ≤ 0 then 0 else 1) + ((if y − 21 ≤ 0 then 0 else 1)
        + ((if y − 25 ≤ 0 then 0 else 1) − (if y − 23 ≤ 0 then 0 else 1))
          − (if y − 19 ≤ 0 then 0 else 1)) − (if y − 9 ≤ 0 then 0 else 1))
            − (if y − 5 ≤ 0 then 0 else 1) + 1,
     (if y − 5 ≤ 0 then 0 else 1) + ((if y − 23 ≤ 0 then 0 else 1)
       − (if y − 25 ≤ 0 then 0 else 1) − (if y − 7 ≤ 0 then 0 else 1)),
     (if y − 9 ≤ 0 then 0 else 1) + ((if y − 19 ≤ 0 then 0 else 1)
       − (if y − 21 ≤ 0 then 0 else 1) − (if y − 11 ≤ 0 then 0 else 1))]
   )
```

$|\ \_ \Rightarrow None)$›
⟨*proof*⟩

lemma *grid_img_defined*: ‹$((predict_{seq\_layer\_l}\ grid.NeuralNet\ xs) \neq None) = (length\ xs = 4)$›
⟨*proof*⟩

lemma *grid_img_defined'*: ‹$(\exists\ y.\ (predict_{seq\_layer\_l}\ grid.NeuralNet\ xs) = Some\ y) = (length\ xs = 4)$›
⟨*proof*⟩

lemma *grid_image*:
assumes ‹$(predict_{seq\_layer\_l}\ grid.NeuralNet\ xs) \neq None$›
shows ‹*the* $(predict_{seq\_layer\_l}\ grid.NeuralNet\ xs) \in \{\ [0, 0, 1],$
$[0, 1, 0],$
$[1, 0, 0]\}$›
⟨*proof*⟩

lemma *grid_image_approx*:
‹*ran* $(predict_{seq\_layer\_l}\ grid.NeuralNet) \subseteq \{[0, 0, 1], [0, 1, 0], [1, 0, 0]\}$›
⟨*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* $(predict_{seq\_layer\_l}\ grid.NeuralNet) = \{a.\ length\ a = 4\}$ ›
⟨*proof*⟩

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

lemma ‹$x3 \in \{0.96..1.00\} \wedge x2 \in \{0.96..1.00\}$
$\wedge\ x1 \in \{0.00..0.04\} \wedge x0 \in \{0.00..0.04\} \Longrightarrow predict_{seq\_layer\_l}\ grid.NeuralNet\ [x3, x2, x1, x0] = Some\ [0, 1, 0]$›
⟨*proof*⟩
lemma ‹$x3 \in \{0.00..0.04\} \wedge x2 \in \{0.00..0.04\}$
$\wedge\ x1 \in \{0.96..1.00\} \wedge x0 \in \{0.96..1.00\} \Longrightarrow predict_{seq\_layer\_l}\ grid.NeuralNet\ [x3, x2, x1, x0] = Some\ [0, 1, 0]$›
⟨*proof*⟩

lemma ‹$x3 \in \{0.95..1.00\} \wedge x2 \in \{0.00..0.05\}$
$\wedge\ x1 \in \{0.95..1.00\} \wedge x0 \in \{0.00..0.05\} \Longrightarrow predict_{seq\_layer\_l}\ grid.NeuralNet\ [x3, x2, x1, x0] = Some\ [0, 0, 1]$›
⟨*proof*⟩
lemma ‹$x3 \in \{0.00..0.1\} \wedge x2 \in \{0.96..1.00\}$
$\wedge\ x1 \in \{0.00..0.1\} \wedge x0 \in \{0.96..1.00\} \Longrightarrow predict_{seq\_layer\_l}\ grid.NeuralNet\ [x3, x2, x1, x0] = Some\ [0, 0, 1]$›
⟨*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*:
‹$\models_{il} \{inputs\}\ (predict_{seq\_layer\_l}\ grid.NeuralNet)\ \{intervals\_of\_l\ 0.000001\ predictions\}$›
⟨*proof*⟩

lemma *grid_meets_expectations_max_classifier*:
‹$\models_{l} \{inputs\_small\}\ (predict_{seq\_layer\_l}\ grid.NeuralNet)\ \{expectations\_small\}$›
⟨*proof*⟩

lemma *grid_min_delta_classifier*:
‹$1.0 \models predict_{seq\_layer\_l}\ grid.NeuralNet$›

⟨*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

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*]:
‹*predict*$_{seq\_layer\_m}$ *grid.NeuralNet* (*vec_of_list xs*) = (*case xs of*
[*x3*, *x2*, *x1*, *x0*] ⇒ *let y* = *2* ∗ (*x3* + (*x2* ∗ *2* + (*x1* ∗ *4* + *x0* ∗ *8*)))
in Some (
vec_of_list[(*if y* − *7* ≤ *0 then 0 else 1*) +
((*if y* − *11* ≤ *0 then 0 else 1*) + ((*if y* − *21* ≤ *0 then 0 else 1*)
+ ((*if y* − *25* ≤ *0 then 0 else 1*) − (*if y* − *23* ≤ *0 then 0 else 1*))
− (*if y* − *19* ≤ *0 then 0 else 1*)) − (*if y* − *9* ≤ *0 then 0 else 1*))
− (*if y* − *5* ≤ *0 then 0 else 1*) + *1*,
(*if y* − *5* ≤ *0 then 0 else 1*) + ((*if y* − *23* ≤ *0 then 0 else 1*)
− (*if y* − *25* ≤ *0 then 0 else 1*) − (*if y* − *7* ≤ *0 then 0 else 1*)),
(*if y* − *9* ≤ *0 then 0 else 1*) + ((*if y* − *19* ≤ *0 then 0 else 1*)
− (*if y* − *21* ≤ *0 then 0 else 1*) − (*if y* − *11* ≤ *0 then 0 else 1*))]
)
| _ ⇒ *None*)›
⟨*proof*⟩

lemma *grid_img_defined_mat*: ‹((*predict*$_{seq\_layer\_m}$ *grid.NeuralNet* (*vec_of_list xs*)) ≠ *None*) = (*dim_vec* (*vec_of_list xs*) = *4*)›
⟨*proof*⟩

lemma *grid_img_defined_mat′*: ‹(∃ *y*. (*predict*$_{seq\_layer\_m}$ *grid.NeuralNet* (*vec_of_list xs*)) = *Some* (*vec_of_list y*)) = (*dim_vec* (*vec_of_list xs*) = *4*)›
⟨*proof*⟩

lemma *grid_image_mat*:
assumes ‹*predict*$_{seq\_layer\_m}$ *grid.NeuralNet* (*vec_of_list xs*) = *Some y*›
shows ‹*y* ∈ { *vec_of_list* [*0*, *0*, *1*], *vec_of_list* [*0*, *1*, *0*], *vec_of_list* [*1*, *0*, *0*]}›
⟨*proof*⟩

lemma *ran_aux*:
assumes ‹∀ *x* . *f x* ≠ *None* ⟶ *the* (*f x*) ∈ *Y*›
shows ‹*ran* (*f*) ⊆ *Y*›
⟨*proof*⟩

lemma *grid_image_approx_mat*:
‹*ran* (λ *x* . *predict*$_{seq\_layer\_m}$ *grid.NeuralNet* (*vec_of_list x*))
⊆ {*vec_of_list* [*0*, *0*, *1*], *vec_of_list* [*0*, *1*, *0*], *vec_of_list* [*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_mat*: ‹*dom* (λ *x* . *predict*$_{seq\_layer\_m}$ *grid.NeuralNet* (*vec_of_list x*)) = {*a*. *length a* = *4*}›
⟨*proof*⟩

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

lemma ‹$x3 \in \{0.96..1.00\} \wedge x2 \in \{0.96..1.00\}$
$\wedge\ x1 \in \{0.00..0.04\} \wedge x0 \in \{0.00..0.04\} \Longrightarrow\ predict_{seq\_layer\_m}\ grid.NeuralNet\ (vec\_of\_list\ [x3, x2, x1, x0]) = Some(vec\_of\_list\ [0, 1, 0])$›
$\langle proof \rangle$
lemma ‹$x3 \in \{0.00..0.04\} \wedge x2 \in \{0.00..0.04\}$
$\wedge\ x1 \in \{0.96..1.00\} \wedge x0 \in \{0.96..1.00\} \Longrightarrow predict_{seq\_layer\_m}\ grid.NeuralNet\ (vec\_of\_list\ [x3, x2, x1, x0]) = Some(vec\_of\_list\ [0, 1, 0])$›
$\langle proof \rangle$
lemma ‹$x3 \in \{0.95..1.00\} \wedge x2 \in \{0.00..0.05\}$
$\wedge\ x1 \in \{0.95..1.00\} \wedge x0 \in \{0.00..0.05\} \Longrightarrow predict_{seq\_layer\_m}\ grid.NeuralNet\ (vec\_of\_list\ [x3, x2, x1, x0]) = Some(vec\_of\_list\ [0, 0, 1])$›
$\langle proof \rangle$
lemma ‹$x3 \in \{0.00..0.1\} \wedge x2 \in \{0.96..1.00\}$
$\wedge\ x1 \in \{0.00..0.1\} \wedge x0 \in \{0.96..1.00\} \Longrightarrow\ predict_{seq\_layer\_m}\ grid.NeuralNet\ (vec\_of\_list\ [x3, x2, x1, x0]) = Some(vec\_of\_list\ [0, 0, 1])$›
$\langle proof \rangle$

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.

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

lemma *grid_img_defined'*: ‹$((predict_{seq\_layer\_m}{}'\ grid.NeuralNet\ xs) \neq None) = (length\ xs = 4)$›
$\langle proof \rangle$

lemma *grid_img_defined*: ‹$(\exists\ y.\ (predict_{seq\_layer\_m}{}'\ grid.NeuralNet\ xs) = Some\ y) = (length\ xs = 4)$›
$\langle proof \rangle$

lemma *grid_image*:
assumes ‹$(predict_{seq\_layer\_m}{}'\ grid.NeuralNet\ xs) \neq None$›
shows ‹$the\ (predict_{seq\_layer\_m}{}'\ grid.NeuralNet\ xs) \in \{\ [0, 0, 1],$
$[0, 1, 0],$

$[1, 0, 0]\}$›
⟨*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.