Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

BachelorThesis

Optimization and Visualization of Neural Networks for Mobile Devices

Can Carlo Arici

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Supervisor:	M.Sc. Johannes Kässinger, Dr. rer. nat. Frank Dürr
Commenced:	September 2, 2021
Completed:	March 2, 2022

Abstract

This work has two goals, foremost the design and implementation of methods to compress a Neural Network. Two experimental compression methods were designed and implemented for this purpose, the first method named Least Significant Path Pruner, calculates the shortest paths from each output node of the Neural Network to any input node. And the pruning of edges common in these paths, by setting the weight to zero until a target sparsity for the neural network is reached. The second method is named Least Significant Node Merger, which merges multiple nodes from a layer into one, until the targeted size is reached. Removing edges and nodes of the network in the process. The second goal of the work is to implement a tool, which can visualize Neural Networks as graphs and features the ability to compare two different Networks against each other. The tool should also be able to display the differences emerging through the compression of the Neural Network. The result is the application "ShowDiff", which combines these two goals into an application, making it possible to compress a model and then compare with the Network it originated from.

German Abstract

Die vorliegende Bachelorarbeit verfolgt zwei Ziele. Das erste Ziel ist, das Konzeptionieren und die Implementierung von Methoden zur Komprimierung von neuronalen Netzen. Hierfür wurden zwei neue Methoden entwickelt, die erste Methode mit dem Namen Least Significant Path Pruner, welcher für jeden Ausgangspunkt des Netzes den kürzesten Pfad zu einem der Anfangspunkt berechnet. Und die Kantengewichte, welche oft in den Pfaden vorkommen gleich null setzt, bis ein vorgenommener Prozentsatz der Kanten null als Gewicht haben. Die zweite Methode ist der Least Significant Node Merger, dieser Algorithmus verschmilzt mehrere Knoten eines neuronalen Netzwerk Schichtes zusammen um die Anzahl der Kanten im Netz zu verkleinern. Zu diesen experimentellen Methoden wurden auch weitere geläufige Komprimierungsmethoden implementiert.

Das zweite Ziel der Arbeit ist die Implementierung eines visualisierungs Tools, welches neuronale Netze darstellt und die Möglichkeit bietet, zwei neuronale Netze miteinander zu vergleichen. Das Tool sollte dazu die Funktion bieten, die Auswirkungen einer komprimierungs Operation auf das Netz darzustellen. Das Resultat der Vereinigung dieser zwei Ziele ist das Programm "ShowDiff", welches die Möglichkeit bietet Komprimierungsmethoden auf ein neuronale Netz anzuwenden und die Resultate dann mit dem originalen Modell zu vergleichen.

Contents

List of Figures

Acronyms

CNN Convolutional Neural Network. 15

CQ Compression Quality. 7, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53

CR Comression Rate. 44, 45, 49, 51, 52

DNN Deep Neural Network. 7, 13, 15

LSNM Least Significant Node Merger. 19, 23, 26, 30, 49, 51, 52, 53, 55

LSNR Least Significant Node Remover. 30, 49, 51, 52, 55

LSPP Least Significant Path Pruner. 7, 19, 23, 24, 45, 46, 47, 48, 49, 55

MP Modification Procedure. 33

MVP Model-View-Presenter. 19

NN Neural Network. 11, 14, 15, 17, 19, 20, 21, 22, 23, 30, 31, 32, 36, 45, 49, 55

PerSiVal Pervasive Simulation and Visualization. 17, 43

RNR Redundant Node Remover. 55

SVD singular value decomposition. 28, 29

UI User Interface. 15, 20, 32, 33, 35

1 Introduction

The versatility of neural networks are simply put remarkable. From identifying objects to predicting outcomes neural networks can handle use cases impossible for conventional methods. Because of this versatility, it only makes sense, that neural networks can have meaningful uses in mobile devices as well. Like supporting the cameras in modern cell phones for even better performance or enable augmented reality simulation on mobile devices. Through it is attached to difficulties, because until this point the main objective of neural networks was being effective and accurate in the task wished to fulfill, but with the limitations of mobile devices efficiency increases in importance. In modern times where [Performance per Watt] begins to out weight Moore's Law in importance to classify our progress in computation units, efficiency has its advantages regarding neural network inference too. The efficiency of a neural network can be improved by decreasing its size or complexity by compressing it into a smaller model. This compression can lead to being able to use the neural network in the first place, because of the original network size being to heavy storage wise. In addition, the higher efficiency of a neural network could lead in the program which it uses, to more responsiveness by shortening the run time of inference or lessen power consumption resulting in longer usages of the program.

All this leads to the conclusion that compression of neural network is a field which has potential and usage in the field of deep learning. Therefore, this thesis has two goals, first to design and implement beside the already existing compression models, new experimental methods with one being a method removing the network flow with the minimum significance and another one being a method to merge nodes of the Neural Network. The second goal is the implementation of a tool to visualize Neural Networks with the intent to compare different Networks and therefore the ability to compare the resulting Neural Network of different compression methods against each other or the original Network. These two goals flow in the application "ShowDiff", which enables the seamless transition between compression and visualization of Neural Networks.

2 Background & Related Work

2.1 Background

2.1.1 Deep Neural Networks

Although the procedure of learning a model is insignificant for this thesis, the structure of a model is. A neural network can be described as a graph with layers, consisting of an input layer, a hidden layer and an output layer. These layers are a set of nodes and these have edges to the nodes of neighboring layers. These edges have an assigned weight from the learning phase. Nodes of neighboring layers are fully connected. A neural network is considered as deep if it has two or more hidden layers. The nodes of the input layer represent the input data, following the status of the nodes the edges propagate the values of the former layer to the next in consideration of the weights of the edges. The prediction is complete when the output layer is reached.
In most scientific papers, deep neural networks are represented as graphs and the layers are either grouped horizontally or vertically, similar to the example in Figure 2.1.[BL16]

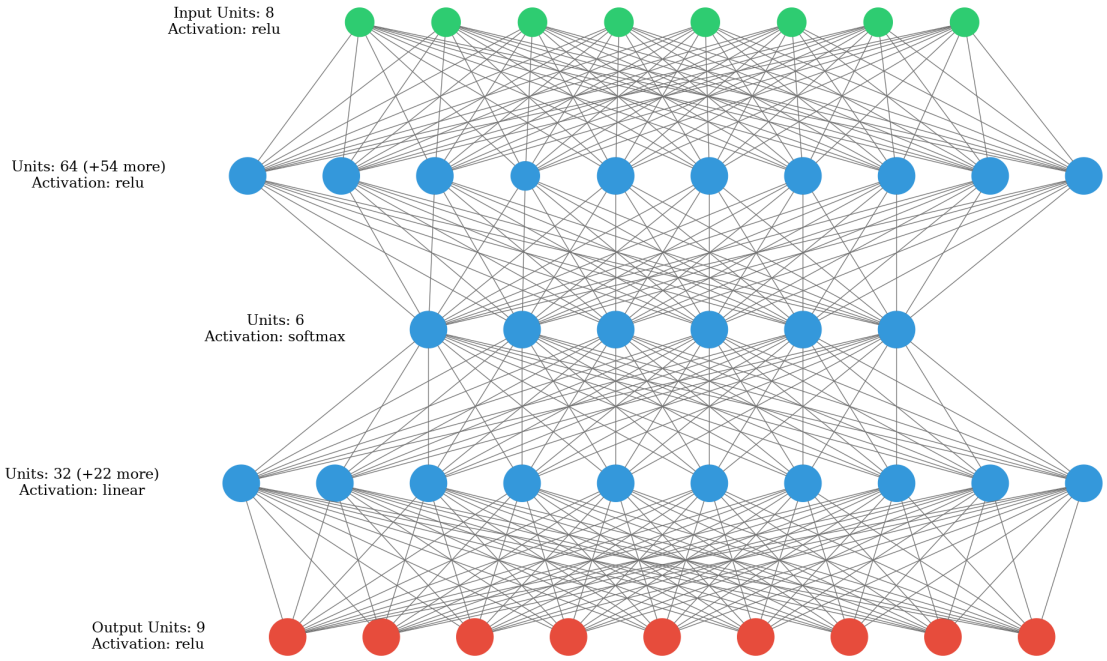


Figure 2.1: Example for a frequently seen DNN representation.[**KerasVis**]

2.1.2 Compression Methods

The complexity of Neural Networks makes it difficult to construct a perfect one that combines the highest accuracy in predictions and the lowest number of parameters needed, it only makes sense that coming up with a perfect Neural Network is viewed similarly to winning the lottery [FC18]. The result is that Neural Networks are mostly over parameterized. The other approach to simply trial and error a good structure for a NN, is to compress an existing one and gaining a simpler and smaller network in the process. In this thesis the compression methods existing are divided into two categories. First, there is the pruning approach, which works by removing edges of Neural Networks by setting them to zero. Pruned edges can be used in further compression steps. Calculations with pruned edges can even be skipped with the use of special sparse layers, leading to time reduction in the inference.[LWF+15] Secondly, the method of reshaping means the modification of the structure of the Neural Network by either removing nodes within a layer or adding new layers to the Neural Network, with the intention to reduce the total number of parameters in the Neural Network. Commonly referenced methods of these categories are among other things, such as iterative weight pruning and weight factorization. Where the former prunes the edges of layers nearest to zero and then trains the network in an iterative manner, does the latter reshape the network by dividing one fully connected layer into two layers approximating the original layer. These two methods were also implemented and will be described in detail in chapter 5 .

2.1.3 Software

Keras

Keras is a Deep Learning API written in python, aiming to be a user-friendly and modular API. Keras runs on top of TensorFlow 2 and features methods to create Neural Network for Machine Learning purposes. Keras also features methods to import and export models in the Keras specific HDF5 file format, models can then be run on different platforms.

Keras features sequential and functional models. The latter, not being of importance in this work, have additional features such as shared layers, non-linear topology and multiple inputs or outputs. A sequential model is a structure of successive layers. Keras features the ability to implement custom layers and provides multiple different layer types, such as the dense layer being a fully connected layer with weight edges and biases, dropout layer randomly setting input values to zero and flatten layer reducing multidimensional inputs to one dimensional outputs. Dense layers can be initialized with different activation functions to describe how the input values are modified, the default being a linear function. Each layer is initialized with a number of units functioning as the output nodes of a layer. Input nodes of the layer is either decided by the output nodes of the former layer or by initializing an input form. Then the model is compiled with a loss function and an optimizer, the former being the metric to optimize, while training, and the latter being the method changing attributes of the model.[Keras]

Bokeh and Panel

Bokeh is an open source Python library under the BSP-License. Enabling interactive visualization of data sets for modern web browsers. It features methods for rendering different structures and objects in plots and enables the user to interact with them in different ways.[Bokeh]

Panel is an open source Python library, which features templates and UI elements, such as to create interactive apps and dashboards. Bokeh plots are supported in Panel and can be displayed in figures. Panel is built on infrastructure provided by Bokeh and is therefore highly compatible with Bokeh.[Panel]

2.2 Related Work

When it comes to related works there are similar areas, which this thesis is related to. First there are programs such as DeepX[LBG+16], which function like a black box to not trouble the user with the inner workings and try to execute compression (or similar methods) of a given Neural Network to enable it to run on mobile devices or devices with limited resources. In addition, there is the DeepX Toolkit [LBM+16] providing the used methods in DeepX. Then there are programs or methods such as in IS-ResNeXt [LS18], which sparsifies the Neural Network, while training to achieve a more compressed version than what would be possible with conventional methods. On top of that TensorFlow and Keras to feature methods and options to compress Neural Network such as pruning weights in the training phase and clustering of weight to reduce the storage needed. And lastly SparseSep[BL16] a program to optimize deep learning model layers of CNN and DNNs.

Beside works on compression, works of the visualization of Neural Networks shall be described as well. GraphVis [GRAPHVIZ] features much visualization on different graph like structures, but the Neural Network visualization only describes the layers. A similar feature is also implemented in Keras. When it comes to visualizing with more details like nodes and edges there is the library Keras Visualizer [KerasVIZ], which unfortunately does not represent the weights of the edges.

In the two aspects of compression and visualization of Neural Networks there are none established works, where both of them are combined. The motion to compare two Neural Networks is nowhere to be found, either the visualization of the impact of compression.

3 Problem-Statement & System Model

Applications benefiting from Neural Networks can be found anywhere from academic uses such as the Pervasive Simulation and Visualization (PerSiVal) project of the University of Stuttgart to private uses, like the improvement of pictures taken by a modern cell phone. But designing a Neural Network for a specific task can be a tedious and time-consuming work, finding the perfect Neural Network is often compared to winning the lottery[FC18]. Therefore, most of the designed Neural Networks tend to be over parameterized, resulting in bigger and more complex Neural Networks than needed. The size and complexity of Neural Networks makes it hard to understand, sometimes compressing is a better choice than designing and training a new one.

Compression of Neural Network (NN)s can lead to smaller and simpler NN leading to faster inference and reduced save space needed. A compression method either the number of edges or the number of nodes of a NN are reduced, while also trying to keep the performance of the NN as close to the original as possible. The resulting NN are a compromise between compression and performance drops.

Visualizations of Neural Networks can help to understand the shortcomings of different networks by giving visual clues of the structure. In addition, the visualization of compressed models can depict the influence of the compression methods to the original model, granting a better understanding of the changes and procedures of the compression methods.

This bachelor thesis has two goals: First, designing and implementation of compression methods, for reducing the complexity of Neural Networks by either pruning or reshaping, to reduce the time needed for inference by reducing the number of parameters and therefore the number of calculations needed for inference. Secondly, an application shall be designed and implemented for the visual comparison of two Neural Network. These being either the original network or networks originating from it. Furthermore, the tool should have the feature to execute the implemented compression methods. The implemented application will be named “ShowDiff”.

A Neural Network is inserted into ShowDiff as an input, the inclusion of training data is optional. The Compression methods implemented can run, without the need for training data sets. The original and compressed NNs can then be visualized. The data sets can be used for the evaluation of the compressed models.

4 Design Overview

This chapter describes the ideas and concepts that went into the design of the compression methods and the application programmed for this thesis. In the following, the new designed experimental compression methods, the application and its features shall be described .

4.1 Compression Methods

In this work, two new methods for the compression of Neural Network were designed. These two are named Least Significant Path Pruner (LSPP) and Least Significant Node Merger (LSNM).

4.1.1 Least Significant Path Pruner

The idea behind LSPP short for Least Significant Path Pruner, is to let the Neural Network structure influence which edges to prune. For this, the Neural Network is interpreted as a graph. In this interpretation the insertion of inputs to propagate into the graph can be seen as a flow. Following this concept the step to remove edges which influence the flow the least, is first to calculate the paths to the outputs with the least influence to the values propagated and removing edges, which are common in these paths. In summary, the method calculates the least significant paths to each output from any input and removes common edges.

4.1.2 Least Significant Node Merger

Least Significant Node Merger is designed as a reshape method with the target to remove nodes between two successive dense layers. The concept is to merge two nodes and approximate the influence of them by only one. Therefore, the least significant nodes will be merged together in an iterative manner until a threshold for the minimum size or max deviation from the original is passed.

4.2 Application

The application was designed with the Model-View-Presenter (MVP) design pattern in mind and is designed to unify both main goals into one. The user can modify an imported model by compressing or training it and visualize the imported or modified models. The evaluation of the models against a data set is also possible. A side goal of the application is, the ability to use the program as a framework for the implementation and comparison of other modification or compression methods.

4.2.1 Pipeline

The application and UI is divided into the Load, Modify and Compare phases. A visual representation is given in the Figure 4.1. In the following, these phases shall be described in more detail. When starting, the application is in the Load phase and transitions to the Modify phase by loading a model. While in Modify phase the user can transition to the Compare phase, this can be done without modifying the model. The Compare phase is divided into the Visualize phase and Evaluate phase. The user can return anytime from the Compare phase to the Modify phase. When returning to Load phase from the Modify phase the application is reset.

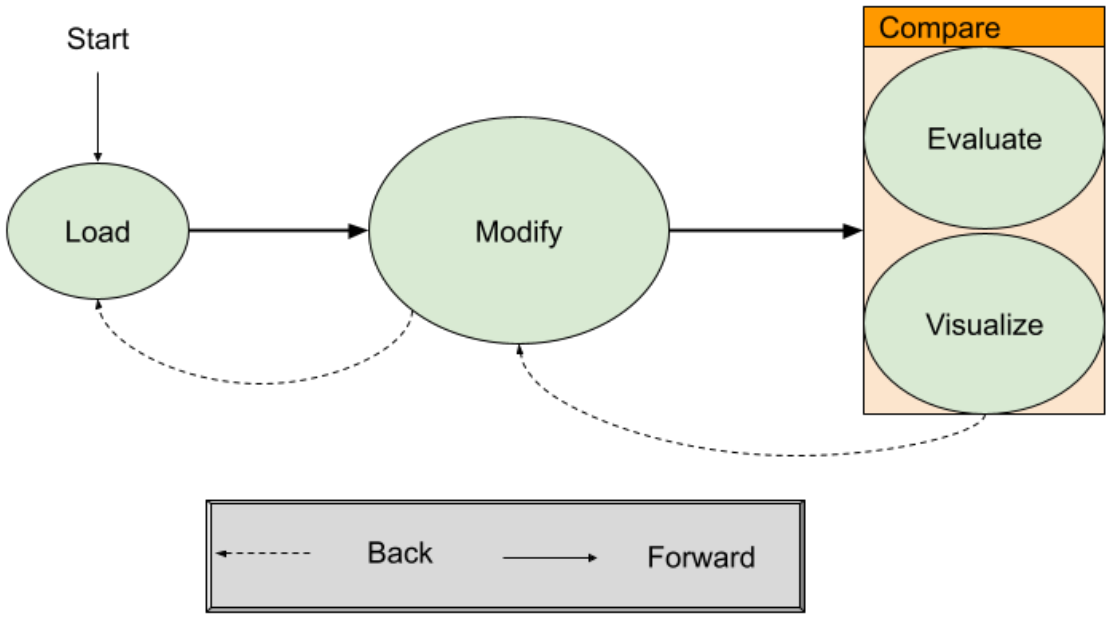


Figure 4.1: Pipeline of the Application

4.2.2 Load Phase

In this phase the user can import models of Neural Network (NN) to use in the application. The user can import only one saved model at once from their local drives.

4.2.3 Modify Phase

The Modify phase is where the main portion of the data modification takes place. It functions like a hub, where the user can return to read different information of the models, more details in chapter 6, currently in the application and interact with them in different ways. The first main goal of the application the manipulation of models is, managed in this phase. Therefore, this phase has the following tasks:

- Modification of Models
  A model is modified by either training or compressing it. Multiple modifications can be executed in succession. The resulting models are then saved.
- Save Models
  Models can be exported for the usage outside the application.
- Display Model Information
  The application displays information of models to the user.

4.2.4 Compare phase

When it comes to the comparison of the models, there are two different methods. The visualization to enable the user to compare the structure and the details of a model. And the evaluation by comparing the performance of inference. Therefore, this phase consists of two phases namely the Visualize phase and the Evaluate phase.

Visualize Phase

In this phase, the user can compare the structure of different models against each other. Chosen models are visualized as figures to enable a visual comparison for the user. The user can compare two models with the following features:

- Figures from Models
  Models are plots inside a figure, which features functions such as drag and zoom to traverse in the figure. Plots consist of a graph describing the NN and description boxes for the layers of the model. These boxes are holding the information of the activation function, in-and output format, name and type of the layer, for each layer of the NN.
- Shared Axis
  Drag and zoom operations of the user are synced between the two figures representing the models, to help the user to compare them.
- Modifiable Figure Layout
  The user can change the width and height of the figures. In addition, the figures can be displayed on top of each other or side by side.
- Display Structural Changes
  If one of the NNs has more layers than the other, the layers will be organized, so that the smaller graph has visual clues to represent this fact. Furthermore, removed nodes of a NN are represented as blank spaces, to show the user the fact of their removal. These Spaces can be ignored if wanted.
- Settings for Plots
  The plot of a figure can be modified, that the graph and description boxes are lined up from left to right or from top to bottom.

- Visualization of Connection
  The edges of the NN are represented in the graph and are color coded to describe their weight. In addition, the user can only choose to display edges with values zero, not zero or both.

Evaluate Phase

In the Evaluate phase different models can be compared on their performance in inference. Then the user will be able to evaluate against a data set and choose different metrics to compare. The results of each model shall be presented with additional information of the model.

5 Design and Implementation of Compression Methods

In the context of this work, several methods have been implemented in the resulting program, which will be explained in the following. Within the implemented methods, the experimental methods are designed in the context of this work, namely the Least Significant Path Pruner (LSPP) and Least Significant Node Merger (LSNM). Beside the experimental methods, multiple additional methods have been implemented to enrich the application with more different methods and approaches. These methods are used as cross-check to evaluate the experimental methods.

The compression methods were implemented entirely in Python. The Python libraries TensorFlow and Keras were used.

5.1 Least Significant Path Pruner

The LSPP is a compression algorithm of the prune category. It uses a graph describing the Neural Network (NN) and the run-time variables *target sparsity*, *max iteration*, *prune percentage* and *K* (for K-shortest paths) as inputs to prune weights of a given model. The algorithm calculates in each iteration the K-shortest paths from each output node to any input nodes and calculates the common edges in these paths, to prune from the model. The algorithm stops when each layer of the model has the *targeted sparsity* or the *max iteration* is reached.

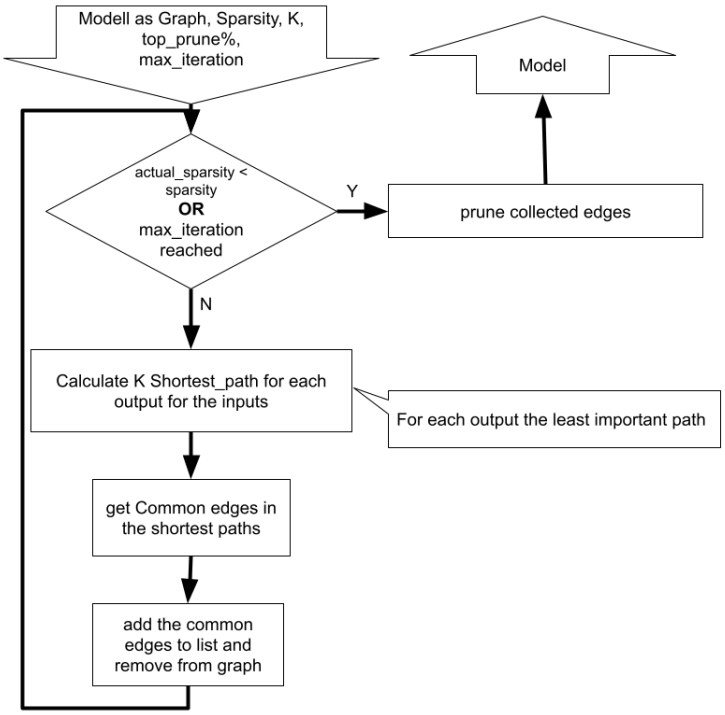


Figure 5.1: Least Significant Path Pruner Flow Diagram

5.1.1 K-Shortest Paths

The first step is to generate the graph by extracting the nodes and edges from the model, by enumerating though the nodes(units) of the model, creating an edge for each input to output node from each layer and linking the corresponding weights to it. Edges with weight values equal to zero are discarded and negative values are replaced by their absolute value.
After this, a modified version of Yen's algorithm is used to calculate the K-shortest paths inside the graph. The algorithm computes single-source K-shortest loopless paths for a graph with non-negative edge cost. The first step is to calculate the shortest path from an output to the inputs by using the Dijkstra algorithms, with the exception that the algorithm does not have to calculate all shortest paths because it terminates when any input is reached. The resulting path is the fastest path from the output to any input. Then this path is used in Yen's algorithm, at each node of the path the next edge is removed and a sub-path is calculated from that node to one of the input nodes as shown in figure 5.2. In the end, the calculated paths are sorted by their path cost and the K-shortest are returned.[Yen71]

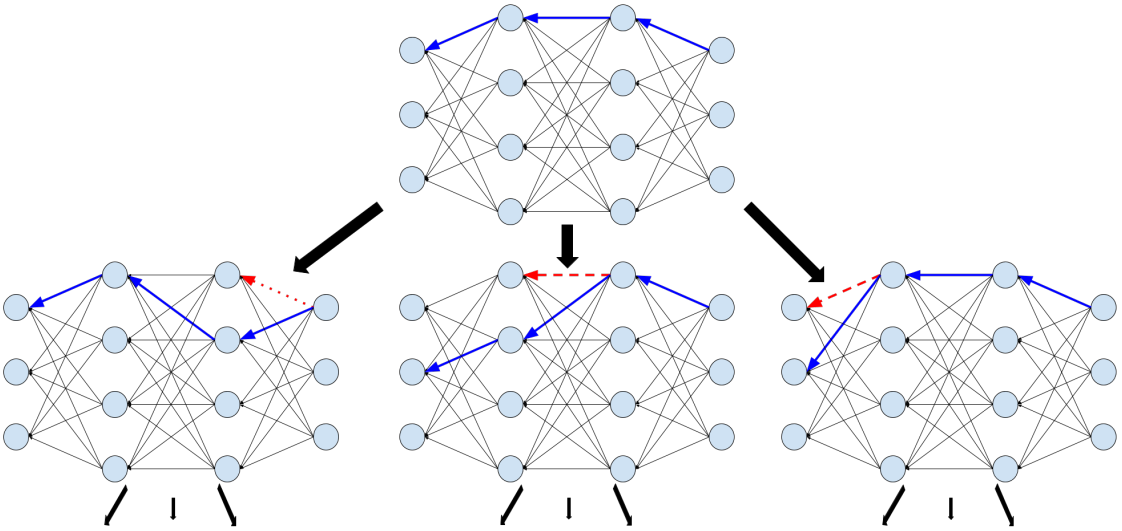


Figure 5.2: Visualization new path calculation Yen's Algorithm

5.1.2 Common Edges calculation

To calculate the common edges of the shortest paths from the last step, a grading system is needed. Therefore, the algorithm grades the paths with points dependent how short they are. The shortest path getting the highest points and every longer path then getting lesser points, so that each path have descending points depending on their rank in shortness. This is done for each paths to output combination.

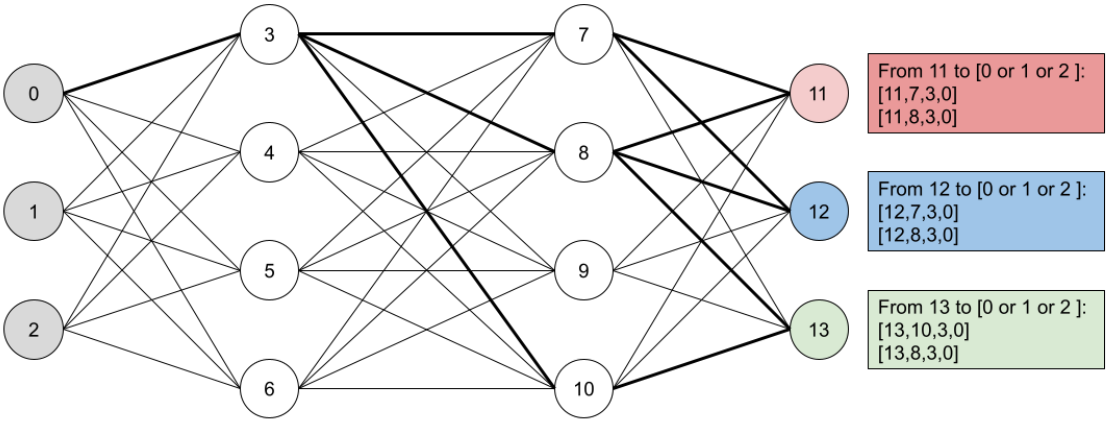


Figure 5.3: LSPP Path Points

Then the paths are divided into edges, with the edges having the points of the paths they originated from. After that, similar edges are combined and their points added together. The results at this moment are a set of edges, with points for each output. Then each edge is then multiplied with the similar edges from other sets. When an edge is absent in a set, a value of 0.5 is used, to get the overall points of an edge. Then the edges are grouped into their layers, where they reside in.

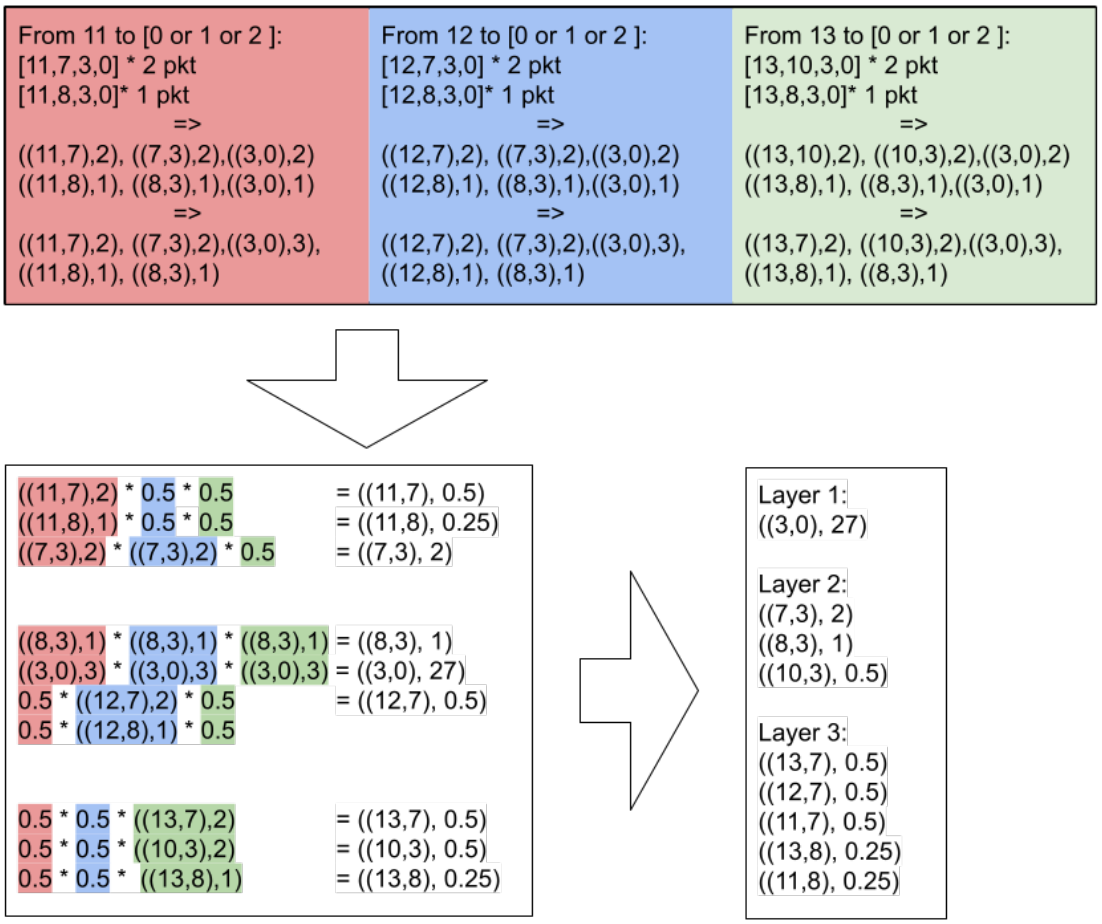


Figure 5.4: Point Calculation

Finally, the edges with the highest points per layer are pruned, the number of top edges to prune is decided by the input variable of *prune percentage* and the difference of the actual sparsity and the *target sparsity*.

5.2 Least Significant Node Merger

Least Significant Node Merger is a compression method of the reshaping category and is one of the experimental Methods designed and implemented in this work. The algorithm has the inputs of *max deviation*, *min size*, *step size* and a *layer pair*. The algorithm merges multiple nodes of a layer into one, the amount depends on the *step size* variable. This is repeated until the *max deviation* between

the original and the result is exceeded or the *min size* of the nodes is reached. The deviation is the difference between the L_2-norm of the original and compressed layer weights. Nodes, which should be merged are calculated after each merging, more will be described in subsection 5.2.1. How the merging is executed is described in the following.

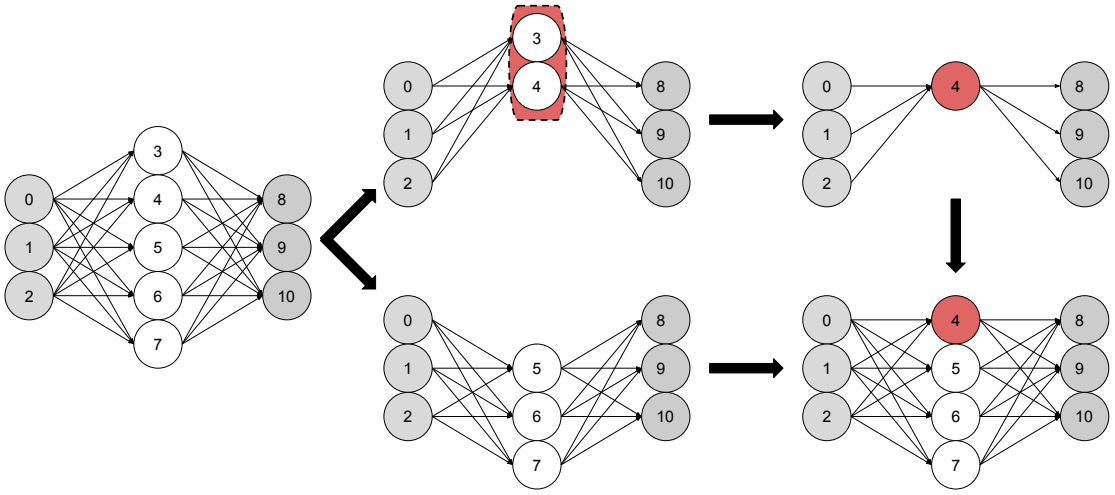


Figure 5.5: Procedure of merging nodes

5.2.1 Candidates

For each node the L_2-norm for their incoming weight matrix and outgoing weight matrix is calculated and the resulting values of each node are multiplied to gain the value of *significance*. Then nodes are the sorted, after their *significance* value and the nodes with the lowest value are picked as candidates for the merging. The number of candidates dependents on the *step size* variable.

5.2.2 Merging

First, the edges from the graph leading and coming from the nodes to be merged are extracted into new matrices, resulting in two distinct pairs of weight matrices. The first pair, describing the influence of the nodes to be merged and the second, being the rest of the original network. The weight matrices of the extracted graph are $u(m \times s)$ and $v(s \times n)$ with s being the *step size*. Visualized in figure 5.6.

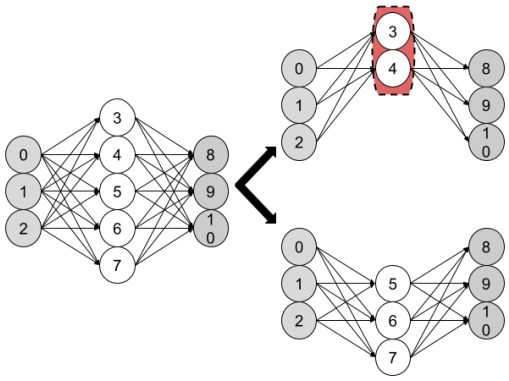


Figure 5.6: Graph splitting

Then these are multiplied, to gain the matrix M, describing how the input nodes of the first layer manipulate the output nodes of the second layer. Then the **singular value decomposition** is used to generate two new matrices(u' and v') approximating M. By removing dimensions until u' has the dimensions $m \times 1$ and v' has the dimension $1 \times n$, the resulting matrices approximate the original nodes input and output matrices.

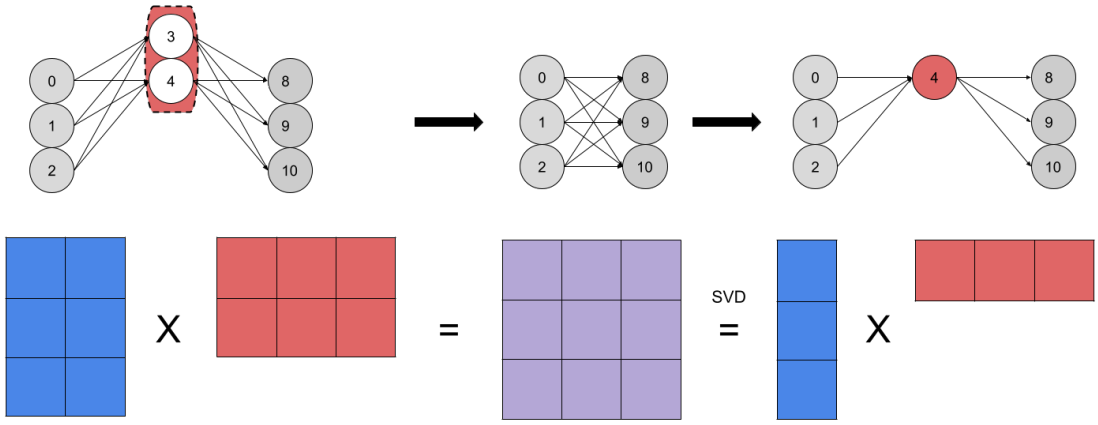


Figure 5.7: Visualization of a node merge step

The merged graph is added to the original graph, to gain the new smaller graph.

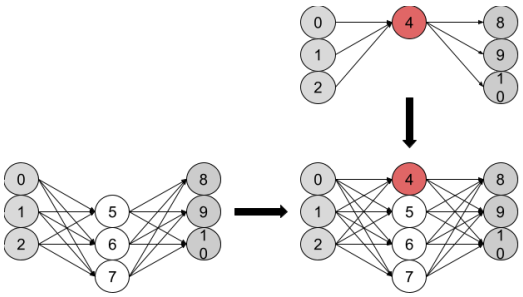


Figure 5.8: Appending the merged graph and the rest of the original graph

Singular value decomposition

The singular value decomposition, is a method for data reduction and can be used to split up a matrix M into $U_{(m\times m)}$, $\Sigma_{(m\times n)}$ and $Vt_{(n\times n)}$, the factorization $U \cdot \Sigma Vt$ results into an approximation of M. Σ is a diagonal matrix, describing how much influence parts of the U and Vt matrices have on the approximation. The number of parameters and values can be reduced by cutting these matrices into the form of $U_{(m\times k)}$, $\Sigma_{(k\times k)}$ and $Vt_{(k\times n)}$, which results in a less accurate approximation with fewer parameters.[XLG13]

5.2.3 Smart Train

To improve the performance of the resulting layers, a method named **Smart Train** can be applied to the resulting layer pair. Therefore, the two original layers and the resulting layers are constructed into Keras models. Randomized input values are generated in the next step, which are predicted by the original layer model, to gain a data set with input and output values. Then the data set is used to train the resulting layer model. The trained layers get extracted from the model and inserted into the final model as result of the method execution.

5.3 Additionally Implemented Methods

In addition to the experimental methods, the following methods were also implemented to be compared against the experimental methods in visualization and performance.

5.3.1 Weight Factorization

Weight Factorization can be applied on dense layers to divide a layer into two, which approximates the original layer. This is archived by using the **singular value decomposition (SVD)** on the weight matrix, describing the original dense layer. With the help of **SVD**, the original $m \times n$ weight matrix results into two matrices with $m \times k$ and $k \times n$. The resulting matrices have fewer parameters, if k is smaller than $\frac{mn}{m+n}$. Following the idea of **Weight Factorization**, the method takes a *model*, the *layers to factorize* and a *max deviation* as inputs. Then the method executes the SVD on the weight matrix $M(m \times n)$ for each *layer to factorize* and minimizes the resulting matrices U and V, so that

U has dimension $m \times k$ and V $k \times n$. The method returns the original matrix as a result. If the subtraction of the L_2-norms of the original matrix and the multiplication of U and V exceeds the *max deviation*, otherwise k is lowered until the *max deviation* is exceeded and the new matrices of the last k under the *max deviation* is used to modify the NN by replacing and inserting the new matrices as layers.[LBM+16]

5.3.2 Redundant Node Remover

The Redundant Node Remover method is a compression method of the reshaping category. The number of nodes and parameters of a model is reduced by removing the nodes which have just outgoing or incoming edges with weight value equal to zero. This method generally does not change the performance of the model, but removes parameters and nodes nonetheless.[HPTD15]

5.3.3 Least Significant Node Remover

Least Significant Node Remover functions similar to Least Significant Node Merger as it also orders the nodes, but cuts the bottom percentage of the nodes and removes them from the resulting model. This method also features the **Smart Train** method used in Least Significant Node Merger.

5.3.4 Keras Pruning

Keras Pruning is a pruning compression method implemented in Keras, which was embedded in the application. It prunes edges under a set threshold in an iterative manner by pruning and training the model until a chosen target sparsity is achieved. Keras Pruning needs training data to execute and has therefore a special place in this work because of this fact.[KerasPruner] Keras Pruning is a similar implementation of the Weight Pruning described in this work [HPTD15].

5.3.5 Simple Pruning

Simple pruning is a simple and straightforward pruning method. Pruning is either done in weight or node mode. The former orders the weights of each layer by their magnitude and prunes the edges with the lowest magnitude until the target sparsity for each layer is reached. The node mode on the other hand, sorts the nodes by the L_2-norm of their outgoing edges and prunes the edges of the least significant nodes.

6 Design and Implementation of Visualization Application

This section describes the implementation of the application mentioned in chapter 4. First the UI layout is described, followed by the data structure of the application and finally the implementation of the phases mentioned in chapter 4. The program was fully implemented in python, with the use of APIs, such as Panel, Bokeh, Keras and TensorFlow.

6.1 UI Layout

The UI is programmed and designed with the help of Panel. For these templates provided by Panel were used. Consisting of a header, collapsible sidebar and main view elements. These function like lists, to hold further layout components like buttons and text fields. The sidebar and main view are overwritten by the presenter, if a phase change is initiated. The structure of each layout and their inner workings of a phase will be further explained in their following sections.

6.2 Data Structure

The Program handles the data inside of dictionaries. Each model imported in the Load Phase and generated in the Modify phase is identified by their name given by the user. For each model the program saves three different entries namely the *Model* object, *Graph Data* and the *Model Info*. The structure and values of these entries are as following:

- Model
  A model of the Neural Network (NN) as a Keras model, the program at this moment only accepts sequential models.
- Graph data
  The data used to describe a NN as a graph and extra values needed in the visualization. These values can be categorized as the following:
  - Layer Values
    Consisting of *layer nodes* as a set of node indexes for each layer. *layer names* as the layer names of each layer of the model, *original layer size* describing the node size of each layer that they had, when first being initialized. And the *layer activations* describing the layer activation functions of each layer.

  - Graph Values :
    node_indexes as a list of indexes of the nodes in the graph and the three values *source*, *target* and *weight*, to describe the connections of the NN as edges. Combined resulting in an edge list format for a graph.

- Model Info
  Values used to describe a model used as parameter in modify methods or to display it in the UI for the user. These values are the compressed model size as *Size in MB*, the number of layers the model has as *Number of Layers*, the ID of the layers that are of type dense as *Dense Layers*, the number of overall parameters as *Number of Parameters*, the zero to non-zero weights percentage of parameters as *Sparsity* and finally the *Net Number of Parameters* describing the number of parameters which are non-zero.

On top of these dictionaries the application also saves the name of the first imported model and important non-changeable overarching model information like the loss function, model optimizer, input-and output shape. These overarching model information are needed if the new modified models have to be compiled again inside the program.

6.3 Load Phase

The presenter starts the program by initializing the Load Phase. Then user has the choice to load a model, by either choosing a model saved under the **user_data** folder located in the program files or by entering the location of the model files. Either way the model is loaded with the **load_model** method made available by the Keras API. When the model is successfully loaded, the dictionaries described in 6.2 are initialized. Furthermore, the loaded model is saved into the *Model* dictionary and a fresh instance of *Graph Data* is generated from the model for the *Graph Data* dictionary. Finally, the presenter switches the UI to the Modify phase. When reentering the Load phase and loading a new model the dictionaries and others are reset.

6.4 Modify Phase

The UI of the Modify state is divided into a sidebar and main view. While the former holds different model and control functions, the letter contains multiple instances of modification procedures.

6.4.1 Sidebar Functions

The user can choose a model from a selection of existing models to interact with. The *Model Info* (6.2) of the selected model is then displayed with the overarching model information under a collapsible card, to inform the user of the characteristics of the chosen model. Furthermore, the user can click on the **Save Model** button to save the model in the **user_data** folder located in the program files, for this the **save_model** method made available by the Keras API is used. To start a **Modification Procedure** the user has to click on **Add Modification**, which adds a new **Modification Procedure** to the main view of the UI.

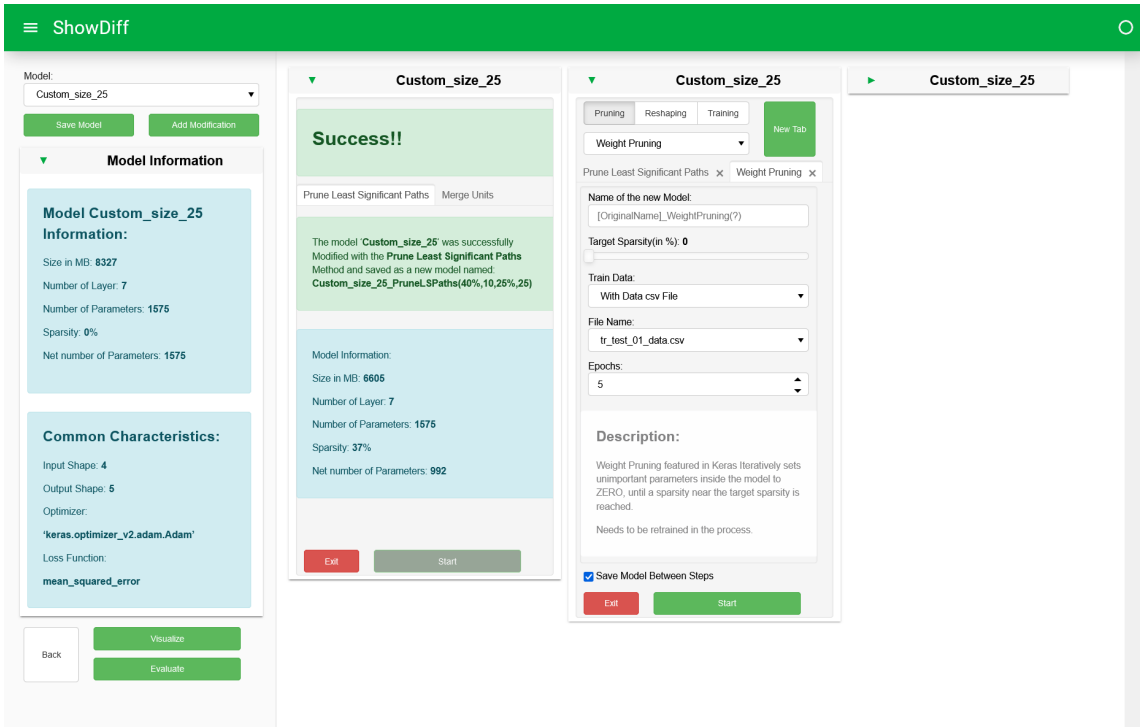


Figure 6.1: UI of the Modify Window
With the sidebar left and from left to right a completed Modification Procedure (MP), a new MP and a collapsed one in the main view.

6.4.2 Modification Procedure

A **Modification Procedure (MP)** is an UI element in form of a collapsible card, the user can select which modification methods to execute. These methods are categorized into pruning, reshaping and training. The user can choose a category to select a specific method from that. The methods available in former two are described in chapter5. In addition, to the one modification method at initialization, the user has the ability to add more methods to execute in succession. Tabs are used to hold the methods, to execute and can be individually removed. The user has the ability to save only the end results or every step as new models. The content of a tab are **modification method boxes** which feature the necessary parameters for the modifications and the new name of the model. When no name is selected, the application generates it by the given parameters and the decided method. When the Modification Procedure starts, the boxes inside the tabs are transferred to a controller method to execute. The originally selected model is then modified in succession alongside the *Graph Data* connected to the model. Then tabs are the actualized with the resulting *Model Info*, after successfully the procedure has terminated. An example how a Modification Procedure looks is given in figure 6.1

Graph Data Modification

When a new model is generated by modifying a model, the *Graph Data* is not generated from scratch like the first model imported in the Load phase. The program rather modifies a copy of the *Graph Data* of the model, the new model originates from. Resulting from the two categories of compression mentioned in 2.1.2, there are three ways the model can change. These are the changing of weights of the model, the addition of layers and the removing of nodes. To ensure the validity of each *Graph Data* the modification of a model is always accompanied by *Graph Data* modification.

Expandable

With the modular nature of the modification method boxes, the addition of new methods to modify models is easy and encouraged. One of the side goals of this application is, the easy comparison of compression methods and making the addition of a new modification method as simple as possible. An user can add a custom modification method by simply adding the code to the program and by assembling a method UI box for the **Modification Procedure**. Accompanying the application is a step-by step tutorial, to follow for the user. This is with the user-friendly and straightforward approach of Panel fairly simple and does not require extraordinary programming skills except python.

Training

The feature to train is added to the modification methods, to enable the user to further improve the new Models resulting from different modification methods. For this, an user can train a model with the train method made available by the Keras API. Then the user can select two different approaches, normal and smart training. With the difference being how the training data is acquired. In the normal variant, the user selects one of the train data files stored in the **user_data** folder. **Smart Train** on the other hand uses the fact that the application has a model of acceptable quality given by the user. By generating training data from the original model, or any other model if wanted, no training data is needed from the user. Training data is generated by first randomizing a set of input values and then predicting the results with the original model. Although the training data generated in this manner is inferior to real training data, it justifies its existence by allowing the user to improve the model, without the need to provide for training data, making the program more independent of extra data.

6.5 Compare Phase

This section describes the implementation of the two different compare phases, namely by visualizing or evaluating .

6.5.1 Visualize Phase

Comparison by visualization is archived by enabling the user to visualize the structure of two models, to comprehend the structural differences.

UI Layout

The UI of the Visualization is divided into the sidebar holding the settings and selections for the model visualization. The buttons to transition back to the Modify phase is also included in the sidebar. The main view holds the figures representing the models. The two models are represented as two figures. When entering the main view, it only displays empty figures and by clicking on **Visualize** they are updated by the plots of the chosen models. Changes in the figure settings are live, which means that no click on **Visualize** is needed. But other changes need a new click on **Visualize** button.

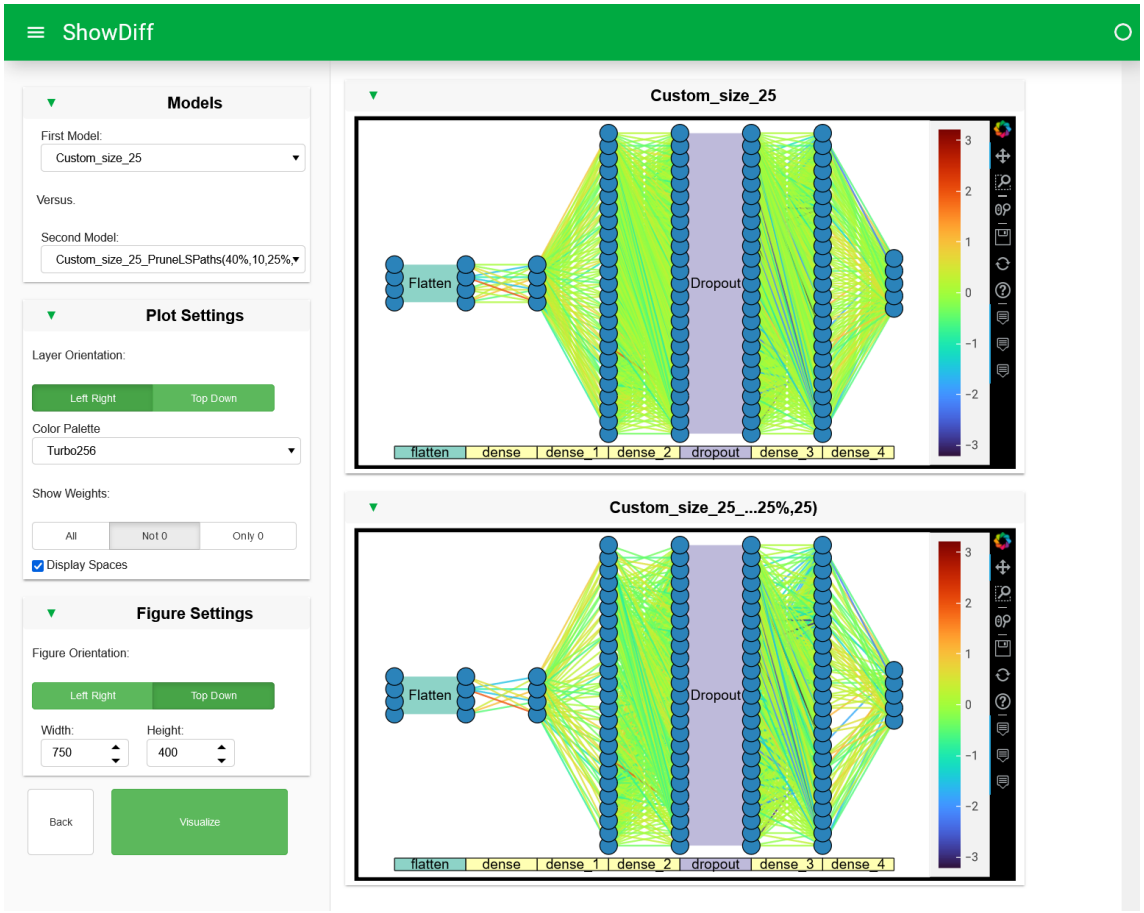


Figure 6.2: UI of the Visualization Window

Axis and Figures

Each model is visualized as a figure, which is a Panel featured UI element. When first initialized the axis of the two figures are synchronized by associating the x-range and y-range attributes of both figures to each other. This has the effect that operations such as drag and zoom are synchronized between the two figures. The figures nested in the main view can be modified by the user to change the width and height and the orientation, meaning if the figures are on top or beside each other.

Rendering the Plot

A model is represented as a plot inside a figure, a model plot consist of a graph to describe the structure Neural Network (NN) and layer info boxes describing each layers information. The plot is entirely rendered by Bokeh. The parts of the plots are visualized by Bokeh's Renderers, which are classes to render specific forms for plots. The layer info boxes are positioned, depending on the plot orientation beside or under the corresponding layer of the graph. Bokeh supports a feature named Hoover Tool, which is used to display information when the user hoovers above a part rendered from a renderer.

The Layer Info boxes are rendered with a glyph renderer and displays the name of the layers. The Hoover Tool for the glyph renderer rendering the layer info boxes displays the name, input- and outputs shape, activation and type of the layer.

The graph renderer from Bokeh is used to render the graph part of the plot. The graph renderer consists of an edge renderer, which render the connections and a node renderer, which render the nodes. When constructing a renderer, a data set has to be given to each of these two renderers. The *Graph Data* explained in 6.2 contains the data used for the renderers. The node renderer has the input value of the *node_indexes*. The node Hoover Tool displays the index of the node. For the edge renderer the *source*, *target* and *weight* values are used, the *source* and *target* lists are used by the renderer to connect the nodes of the node renderer with edges. Then the weight values used to define the color of the edges. This is achieved by using a color transformer, which maps the weight value of an edge to a color value, of a color set the user can choose in the settings. Layers of the NN which are not dense layers are represented as colored boxes with the type of the layer written in the middle. The glyph renderer is used for this like for the Layer Info boxes. The edge Hoover Tool displays the information from where to where an edge goes with the weight value it has. The node Hoover Tool displays the index of the node.

Unit Positions

The positions of the nodes of the graph are randomized, when the renderers are initialized. To order and organize the positions of the nodes inside the figure a layout provider is needed. This layout provider requires the x and y coordinates of each node inside the figure. Elements in a figure are displayed in a coordinate system, therefore the positions have to be calculated. The unit position enables the feature rotate of the plots inside the figure and to display two models with different layer

counts, implied that they originate from a common model. For this first the *Layout Map*s of the models has to be calculated and then the coordinates of the nodes with the help of it.

*Layout Map*s are used to describe where the layers of the model is located, with them “empty“ layers can be added to imply that the other displayed model has an inserted layer in this position. The algorithm uses the saved *layer names* in the *Graph Data* for this.

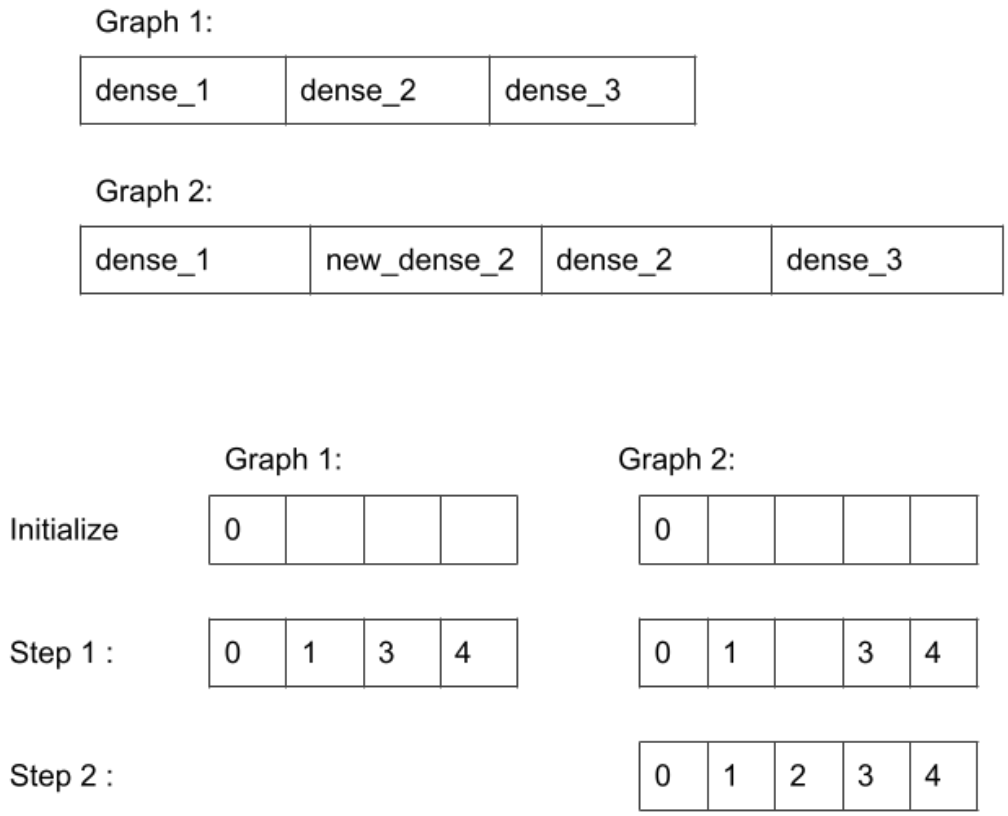


Figure 6.3: Calculation of a Layout map

In the figure 6.3 an example is given. The layout map is designed as an array to indicate the position of the layer in a space. For each graph, an array is initialized with the first entry being 0. Then a list of *common names* is created with the names existing in both graphs. Then for each value in *common names* the *Layout Map* of both graphs are set by the number of unique names that came before the common name in both graphs plus one. Finally, the entries of *Layout Map*s without entries are filled by using the entries before plus one.

The dimension of the graphs are given by the max node count of any layer as *max_nodes* and the last entry of the layout map as number of displayed layers.

Different parameters influence the final calculation of node position. The node positions ares saved as dictionary with the node *ID* as key and the value of the coordinates as a tuple.

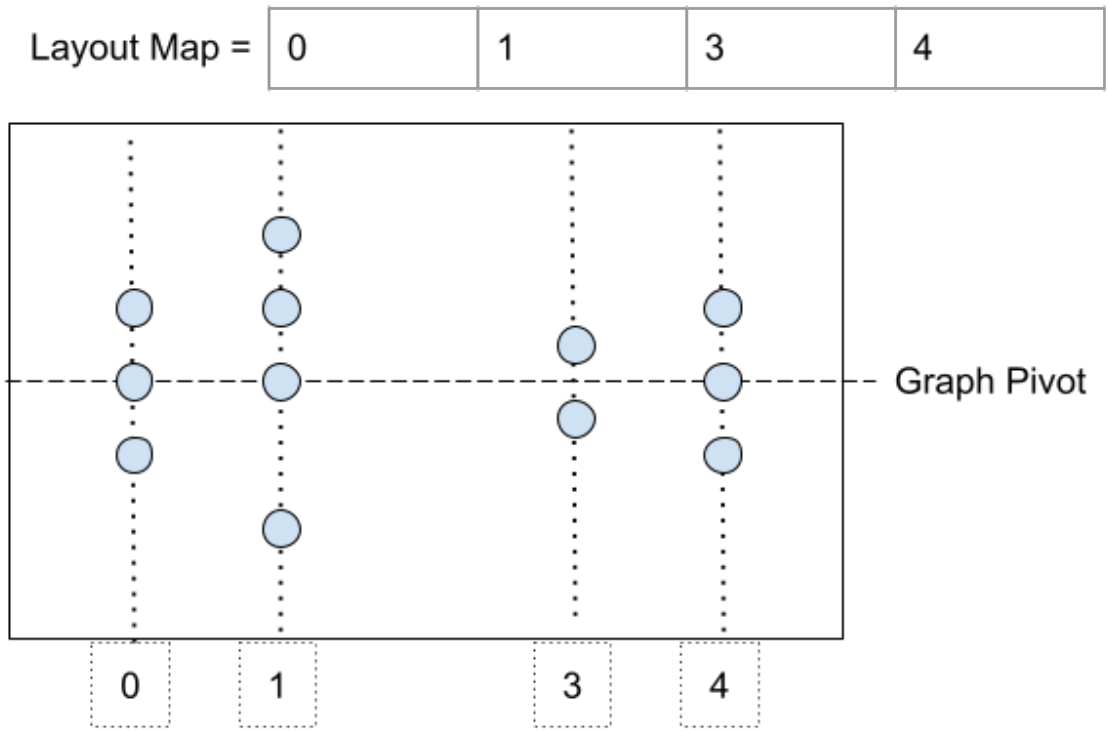


Figure 6.4: Visualization of the graph layout

Figure 6.4 displays an example of the graph layout in the left to right orientation, the *Layout Map* indicates where the layers are in the x-axis displayed as pointed lines. Then a graph pivot is appointed, which describes the line where the middle point of each layer should be. The nodes of a layer are set on their pointed layer lines in a manner that, the pivot line is in the middle of the nodes of the layer. Removed nodes of a graph can be displayed when wanted, by skipping the position where it should have been, as displayed in the second layer of the figure 6.4. If the graph is visualized in the top to bottom orientation, the layout is simply rotated so that the first layer line is on the left top side and the graph pivot runs from the top to the bottom instead.

Settings

- Plot Orientation
  Rotates the plots inside the figures so that the input layer is at the top and output layer at the bottom. This slightly modifies the node position generation.
- Show Weights
  Filters the edges of the graph so that only the selected type of edges such as showing all edges, do not show edges with value equal to zero or only show edges with weight equal to zero are shown.

- Color Palette
  Selection of different color palettes to choose the colors the edges are colored in.
- Display Spaces
  If the nodes removed should be displayed by inserting an empty node where they should have been or not.
- Figure Orientation
  Decides if the figures displayed in the main view are on top or beside of each other.

Examples

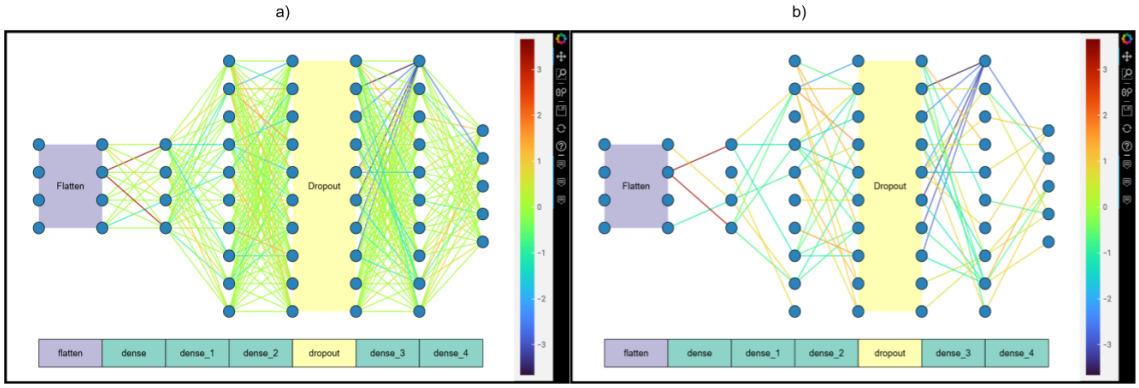


Figure 6.5: Example of pruned edges
Example of a pruned model to show the **Show Weights** a) is with “all“ and b) is with “not 0“.

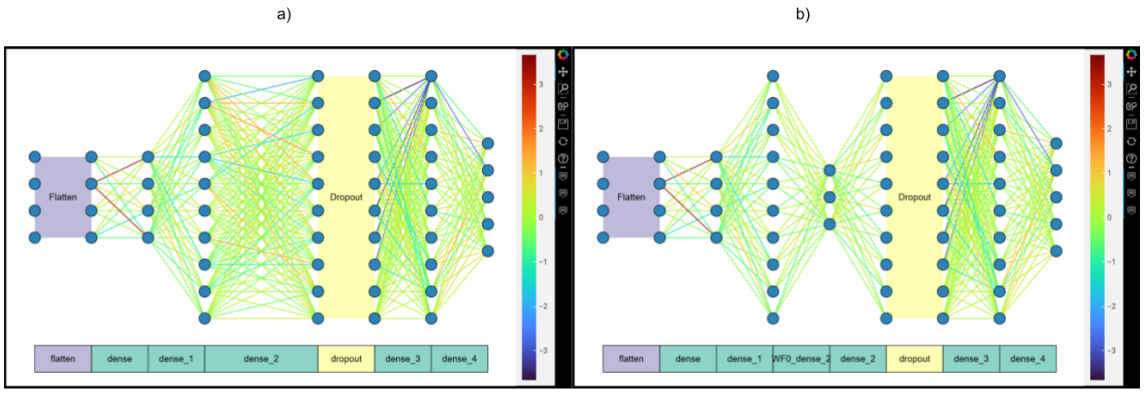


Figure 6.6: Example of models with different layer counts

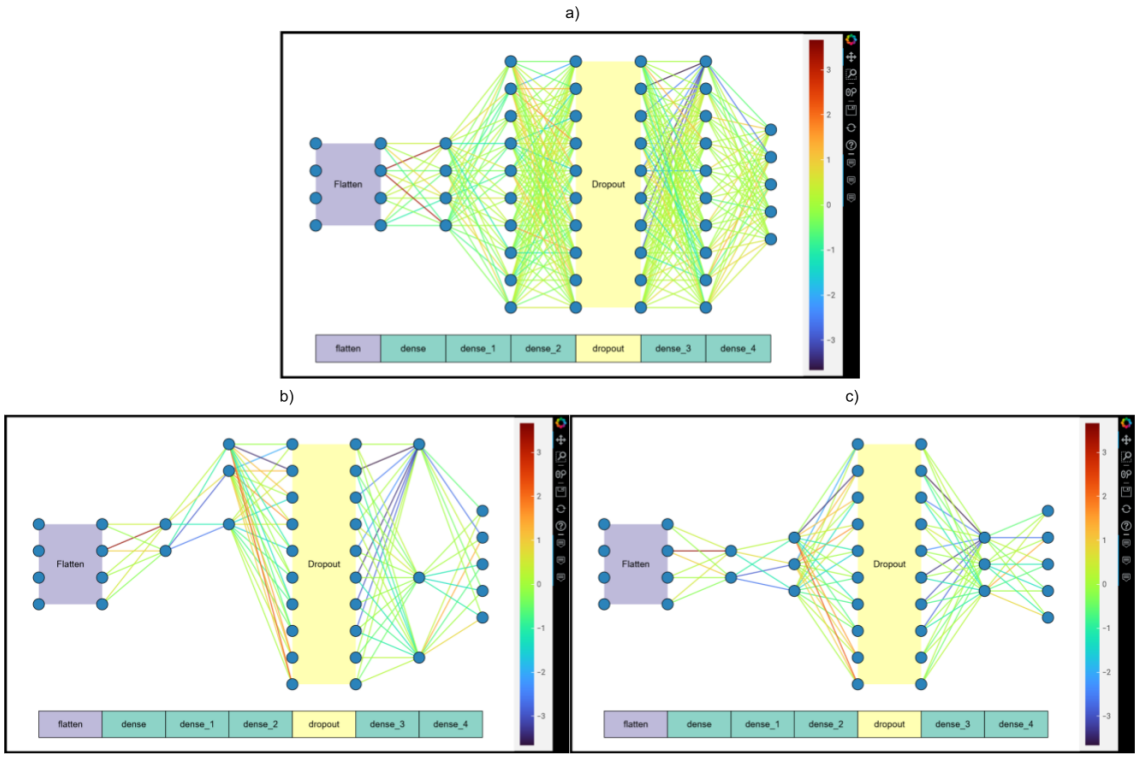


Figure 6.7: Example of models with removed nodes
Example of a graph where nodes were removed, a) original model, b) is a model with removed nodes with spaces where the removed nodes were and c) as a model with removed nodes where the spaces were removed.

6.5.2 Evaluate Phase

The sidebar in the Evaluate UI holds the selection which models to evaluate, the metrics to evaluate and which data to use for the evaluation. The main view shows the results of the evaluation, with the chosen metrics and the information of the graph saved in the model info. The Evaluation is executed by using the method **evaluation** made available by the model class of Keras. The chosen evaluation data is used for this.

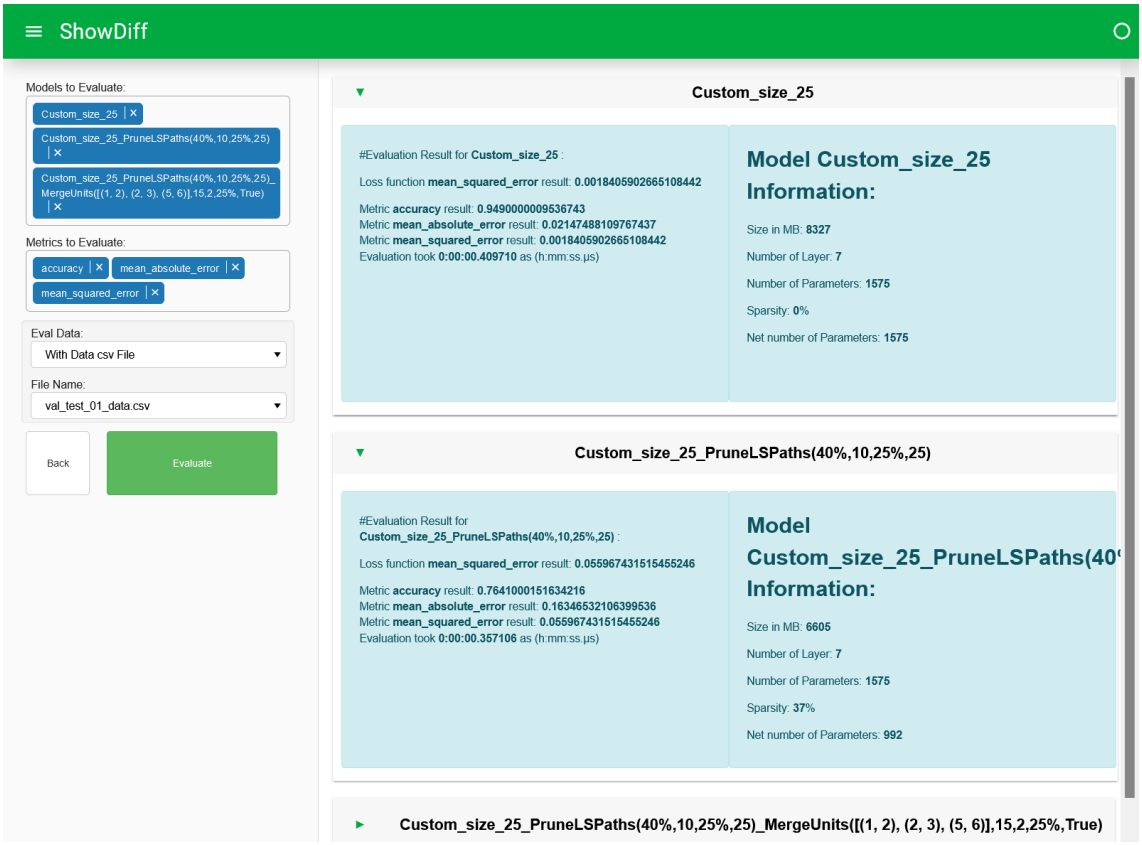


Figure 6.8: UI of the Evaluation window

7 Evaluation

In this section the experimental methods are compared to simpler and more straightforward methods.

7.1 Evaluation Strategy

The Evaluation in this work is divided into comparison of the pruning and reshaping methods against one of the other implemented methods and the evaluation of the influence of the compression types for the prediction run-time for different models. The used models are described in the 7.1.2 section and the metrics for comparison used to compare methods in 7.1.3.

7.1.1 Evaluation Environment

A PC is used for the evaluation. The operating system is Windows 11. The specifications of the PC are CPU as an AMD Ryzen 7 2700X with 8c/16t and a frequency of 4.1GHz, 16 GB of DDR4 Ram and a GeForce RTX 2060 as GPU. TensorFlow2 was run in CPU mode.

7.1.2 Data Sets

Eight different models were created for the evaluation. Data sets from the PerSiVal project were used to train these models. The structure of the trained models were chosen to broaden the diversity of models evaluated. The different models help to determine the influence of different layer sizes, non-dense layers, non-default activation functions and additional layers on the evaluation.

- Custom_25:
  A variation of a model used in the Pervasive Simulation and Visualization] project, with a flatten layer as first followed by dense layers in unit size 4, 25, 25, then a dropout layer and finally two dense layers with 25 and 5 units. Dense layers with 25 Units use the Rectified Linear Unit activation function(ReLU).
- Custom_50:
  Similar to Custom_size_25 with the difference that layers with 25 units have 50 units.
- Full_Dense_25 and Full_Dense_50:
  Similar to Custom_size_25 and Custom_size_50, but with the difference that the flatten and dropout layers are removed.
- Full_Dense_5_25 and Full_Dense_5_50:
  Full_Dense_25 and Full_Dense_50 with two dense layers added in the mid-section.

- Dense_Actless_25 and Dense_Actless_50:
  Full_Dense_25 and Full_Dense_50 with every layer having default activation function.

Table 7.1: Table of used Models

Name	No. Layers	No. Parameters	Layer Types	Layer Units
Custom_25	7	1575	Flatten, 3xDense, Dropout, 2xDense	4,4,25,25,25,25,5
Custom_50	7	5625	Flatten, 3xDense, Dropout, 2xDense	4,4,50,50,50,50,5
Full_Dense_25	7	1575	5xDense	4,25,25,25,5
Full_Dense_50	7	5625	5xDense	4,50,50,50,5
Full_Dense_5_25	7	2875	7xDense	4,25,25,25,25,25,5
Full_Dense_5_50	7	5625	7xDense	4,50,50,50,50,50,5
Dense_Actless_25	7	1575	5xDense	4,25,25,25,5
Dense_Actless_50	7	5625	5xDense	4,50,50,50,5

7.1.3 Evaluation Metrics

Different metrics are needed to describe the overall performance of compression methods. The metrics used in the evaluation are as follows:

- *Compression Rate* (as CR):
  Describing by how much the number of parameters have changed. Compression rate encapsulates both *Reduction* as actual difference in form of removed edges and *Sparsity* as pruned edges. Both of these metrics are percentage numbers. And are calculated as

$$CR = 1 - \frac{\#Edges_{current}}{\#Edges_{original}}$$

- *Accuracy(ACC)*:
  The resulting accuracy of the model tested against the evaluation data set.
- *Compression Quality (CQ)*:
  The metric to describe the overall performance of a compression. Described as:

$$CQ = \frac{Q}{1 - CR}$$

  with Q as $\frac{ACC_{new}{}^2}{ACC_{old}{}^2}$. The result of the function decreases when ACC_{new} decreases and increases with a higher CR value. Because an original graph with Accuracy equal to zero and a *Comression Rate* of 100 percent is unlikely, the value of CQ cannot reach zero. The original model has the *Compression Quality* value of one, higher values represent better quality and values under 1 as a bad compression.

- *Potential* :
  Following the calculation of *Compression Quality* a maximal value as *Potential* can be calculated by setting the value of Q to one and entering a compression rate. The resulting value represents the best value of *Compression Quality* for a given *Comression Rate*, under the assumption that a compression does not improve the model. The metric *Potential* describes the perfect compression where no quality is lost.

7.2 Pruning Evaluation

In this section the models were compressed by using the Least Significant Path Pruner and the Simple Pruner to compare the quality of both compression methods against each other in different Neural Network structures. Each model has two diagrams, first describing the accuracy to sparsity ratio and second describing the Compression Quality to sparsity ratio. Each model was pruned by simple pruner with different target sparsity and from the Least Significant Path Pruner with different parameters. Then the resulting models were evaluated against the validation data set and when two models had the same sparsity the better one was chosen as representation.
The parameters used were:

- Simple Pruning
  With Sparsity = [0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8]
- Least Significant Path Pruner
  Every combination of *max_sparsity* = [0.25; 0.5; 0.75], *max_iterations* = [40], *prune_top* = [0.1; 0.25; 0.5] and K = [5; 10; 15; 20; 25].

In the accuracy diagrams each point represents a model and the Compression Quality diagrams display the *Potential* of the compression rate as black squares and models as points. The Compression Quality (CQ) value of a model count as good, if the CQ is near or equal to the *Potential* of the Comression Rate the model has.

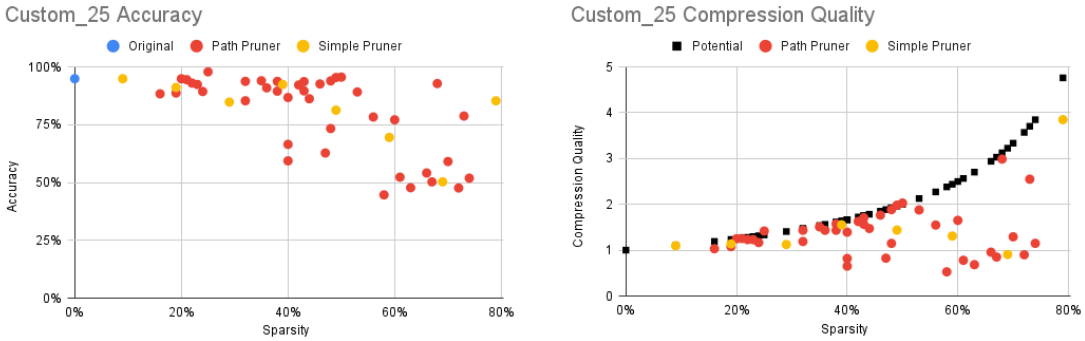


Figure 7.1: Accuracy and CQ Diagram for the model Custom_25

In the diagrams in figure 7.1 can be seen, that the compression methods perform both similarly good, for sparsity under 50%, with the LSPP performing slightly better. Then the CQ of compression deviate strongly from the *Potential* markers.

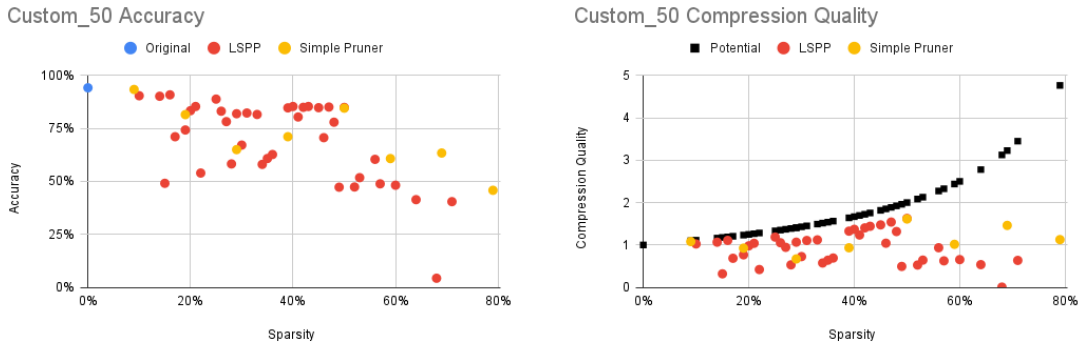


Figure 7.2: Accuracy and CQ Diagram for the model Custom_50

In the diagrams in figure 7.2 can be seen, that both pruning methods perform relatively good for sparsity under 50% with the Least Significant Path Pruner being slightly better in average. The CQ and Accuracy drops significantly for sparsity over 50%.

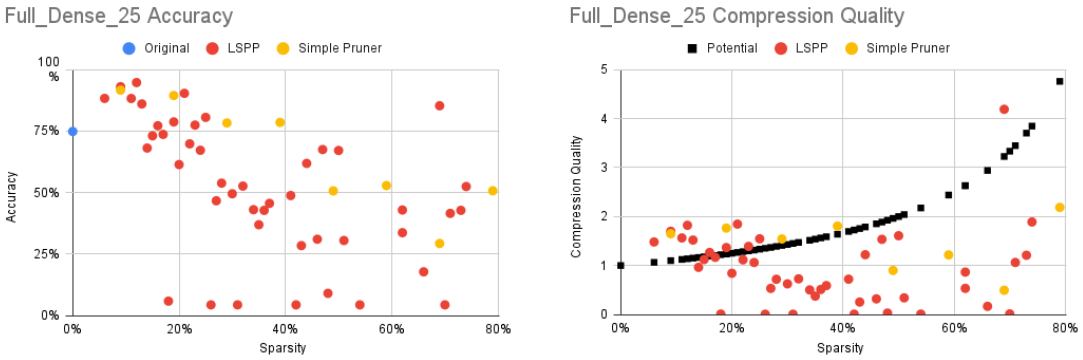


Figure 7.3: Accuracy and CQ Diagram for the model Full_Dense_25

In the diagrams in figure 7.3 an anomaly can be seen in the compression of the model, where for models with sparsity under 30% the Accuracy and CQ extends the accuracy of the original and the *Potential*, the result is that the models exceeding the original are in every way superior to the original, when it comes to accuracy and compression rate. Beside this the pruning methods oscillate with being better than each other.

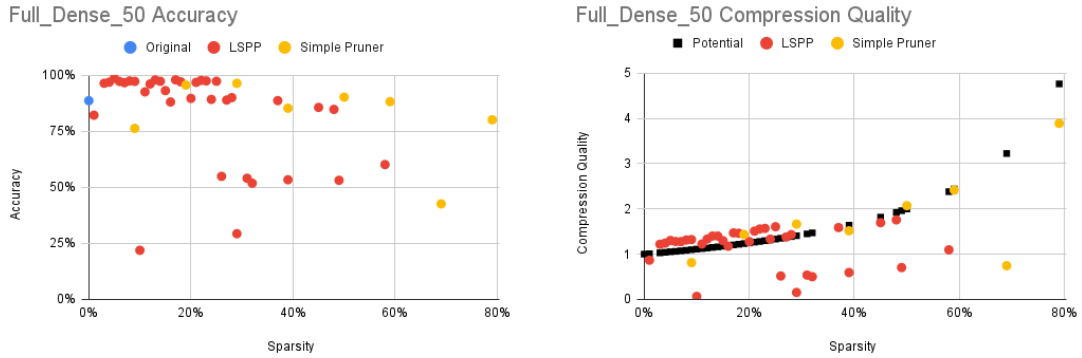


Figure 7.4: Accuracy and CQ Diagram for the model Full_Dense_50

In the diagrams in figure 7.4 can be seen, that the Least Significant Path Pruner method has difficulties reaching sparsity over 60% with the execution parameters. The results are for both methods near their *Potential*, with few runaways.

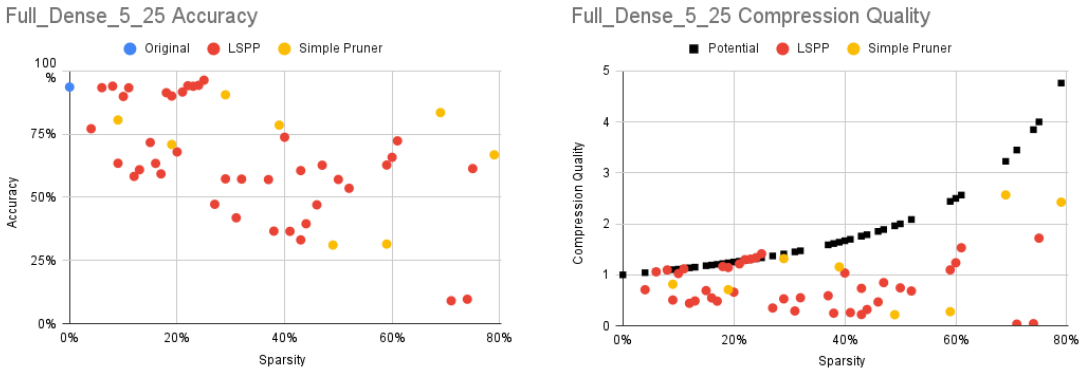


Figure 7.5: Accuracy and CQ Diagram for the model Full_Dense_5_25

In the diagrams in figure 7.5 can be seen, that the results are moderate until 30%, dropping in quality after.

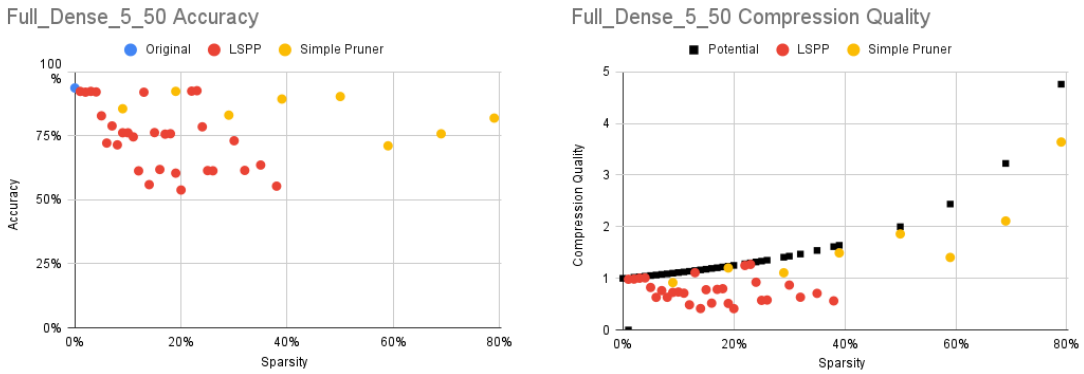


Figure 7.6: Accuracy and CQ Diagram for the model Full_5_Dense_50

In the diagrams in figure 7.6 can be seen, that the LSPP method has difficulties reaching sparsity over 40%. The Single Pruner performs good continuous with the LSPP having single good results in the low sparsity range.

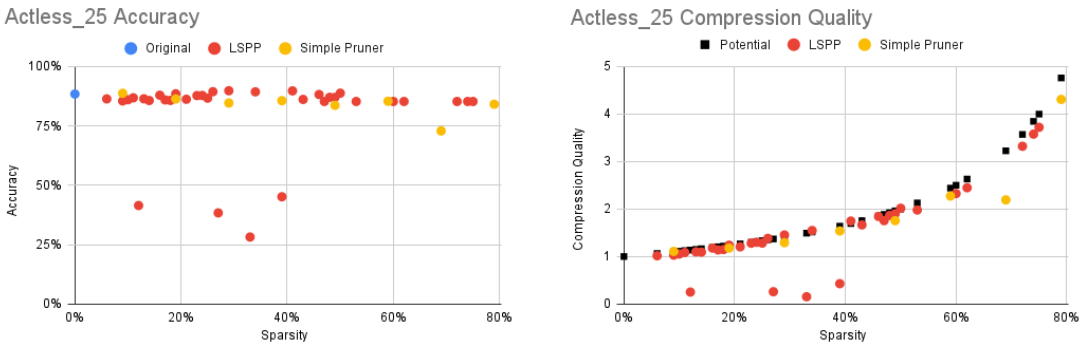


Figure 7.7: Accuracy and CQ Diagram for the model Actless_25

In the diagrams in figure 7.7 can be seen, that the results are except for a few runaways nearly perfect, with the CQ being nearly in equal to the *Potential*.

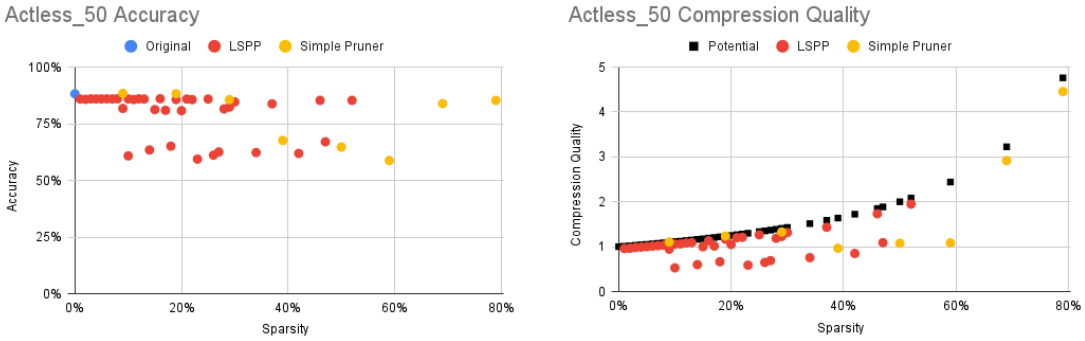


Figure 7.8: Accuracy and CQ Diagram for the model Actless_50

In the diagrams in figure 7.8 can be seen, that the LSPP method has again difficulties reaching sparsity over 60%. The results for both methods nearly perfect for sparsity under 20% and moderate after.

In Summary, the diagrams show that Least Significant Path Pruner has difficulties with large Neural Network (NN), which results in low sparsity. Large NNs lead to more distinct shortest paths, which leads to fewer common edges. Calculating a higher number of paths(K), higher *max iteration* or a more aggressive pruning percentage could lead to higher sparsity, but also increase the time needed and the performance of the resulting NNs. Furthermore, it performs in average slightly better than Simple Pruner in low sparsity(under 40%). The best results are achieved in the "actless" models indicating that non-default activation layers have a negative impact on the performance of LSPP.

7.3 Reshaping Evaluation

The Experimental Reshaping method Least Significant Node Merger is compared with Least Significant Node Remover in this section. Where LSNR removes units deemed least significant, the merge units combines them. Each model was reduced by both methods with different *layer pairs* and *min sizes*, *max deviation* for LSNM was ignored. The different *min sizes* used were 25, 50 and 75 percent and the layer pairs was either all Dense layer pairs or each layer pair individually. Then the resulting models were evaluated against the validation data set and the Comression Rate was calculated. In case of models with equal CR the one with a higher *Accuracy* was chosen. The methods were executed without **Smart Train**(described in 5.2.3).

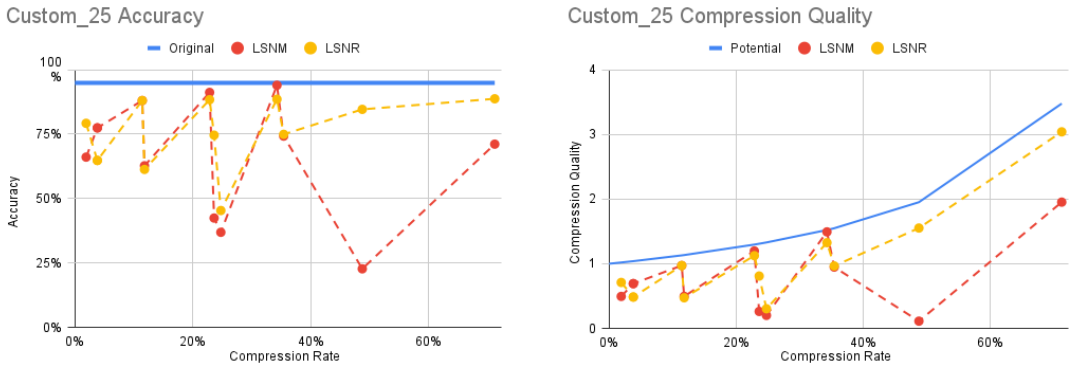


Figure 7.9: Accuracy and CQ diagram for the model Custom_25

In the diagrams in figure 7.1 can be seen, that both methods oscillate between being near and far from the *Potential* line.

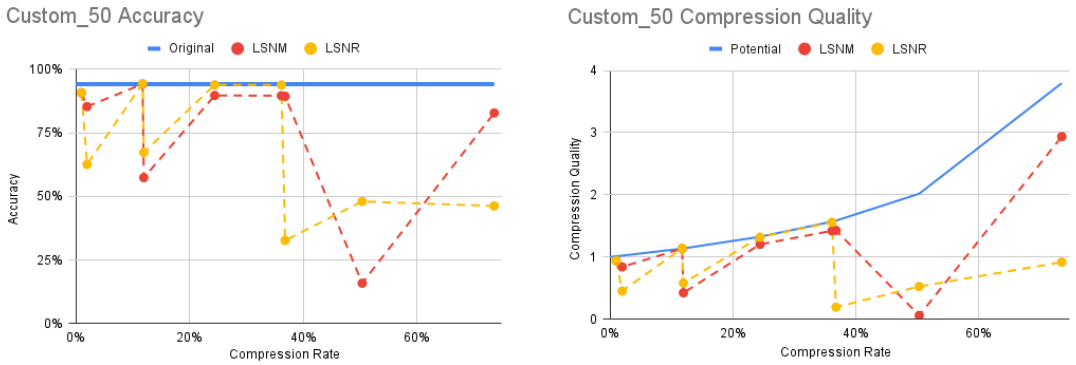


Figure 7.10: Accuracy and CQ diagram for the model Custom_50

In the diagrams in figure 7.2 can be seen, that both methods oscillate between being near and far from the *Potential* line.

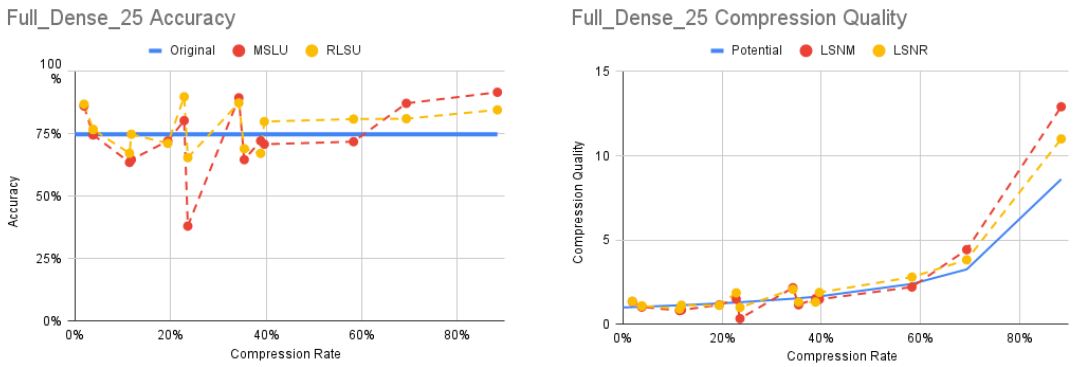


Figure 7.11: Accuracy and CQ diagram for the model Full_Dense_25

In the diagrams in figure 7.3 can be seen, that the Compression Quality of both methods are close to the *Potential* line until exceeding their *Potential*.

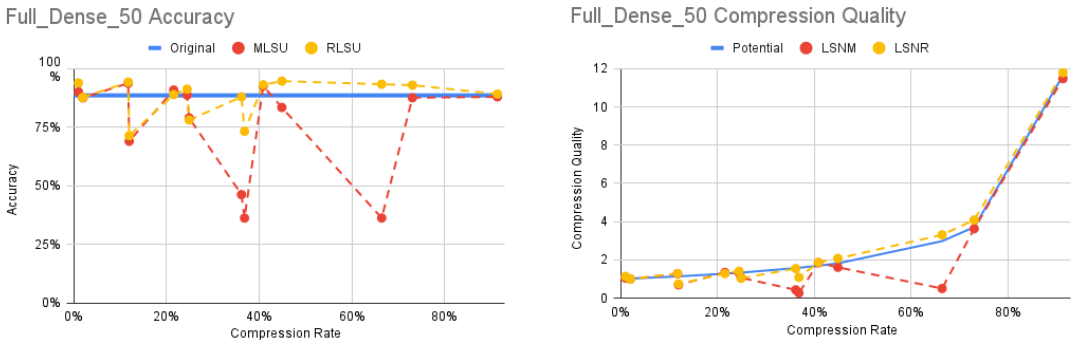


Figure 7.12: Accuracy and CQ diagram for the model Full_Dense_50

In the diagrams in figure 7.4 can be seen, that acrlonglsnr keeps being near the *Potential* line and Least Significant Node Merger oscillate between being near and far from the *Potential* until finally closing up to LSNR at 70% Comression Rate.

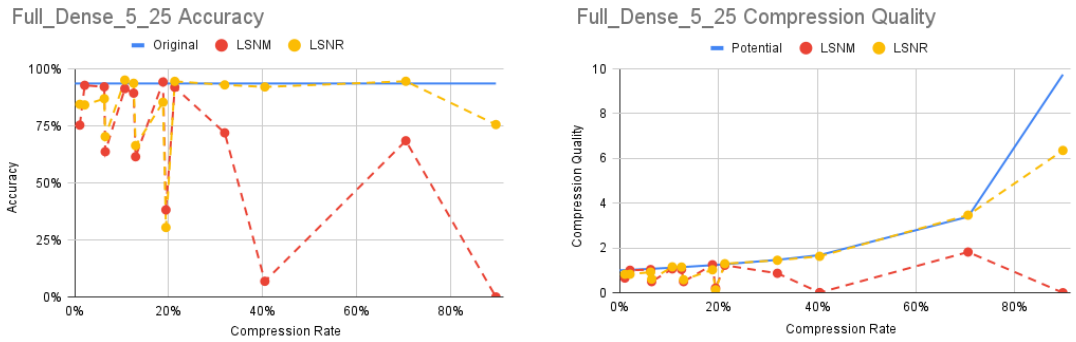


Figure 7.13: Accuracy and CQ diagram for the model Full_Dense_5_25

In the diagrams in figure 7.5 can be seen, that both methods oscillate between being near and far from the *Potential* line until 30% Comression Rate where Least Significant Node Merger deviates stronger from the *Potential*.

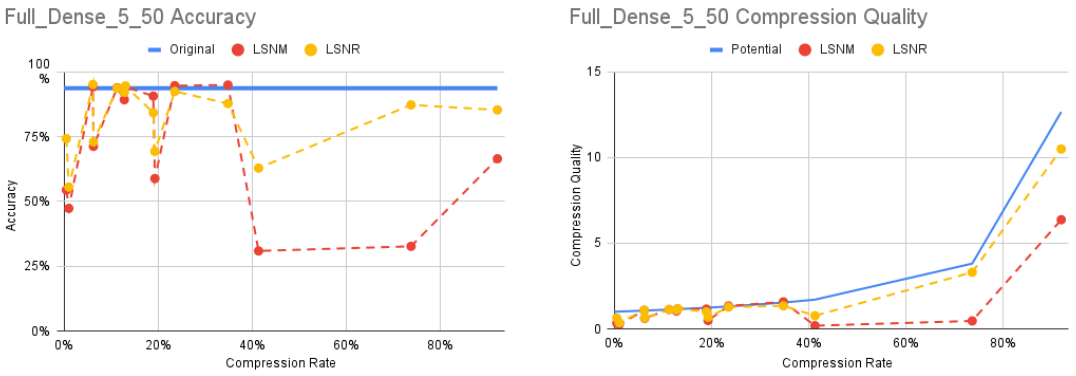


Figure 7.14: Accuracy and CQ diagram for the model Full_5_Dense_50

In the diagrams in figure 7.6 can be seen, that both methods oscillate between being near and far from the *Potential* with LSNM having higher distance to the *Potential*.

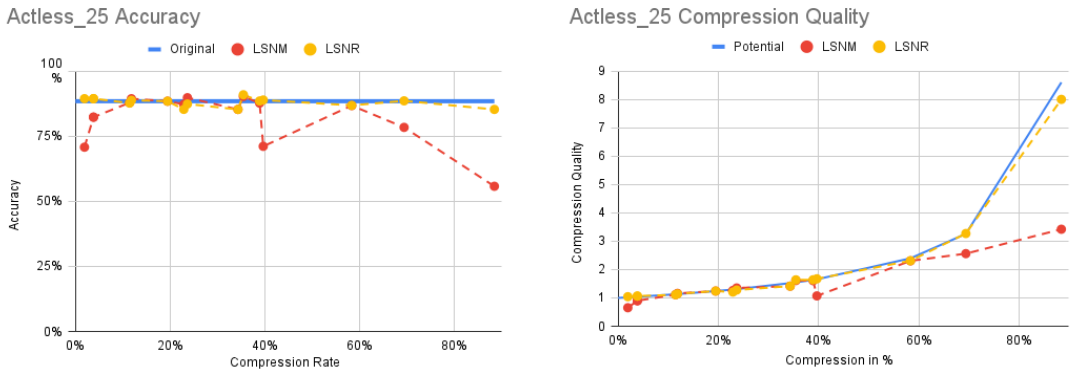


Figure 7.15: Accuracy and CQ diagram for the model Actless_25

In the diagrams in figure 7.7 can be seen, that both methods perform excellent being near the *Potential* until LSNM deviates at 70% Comression Rate.

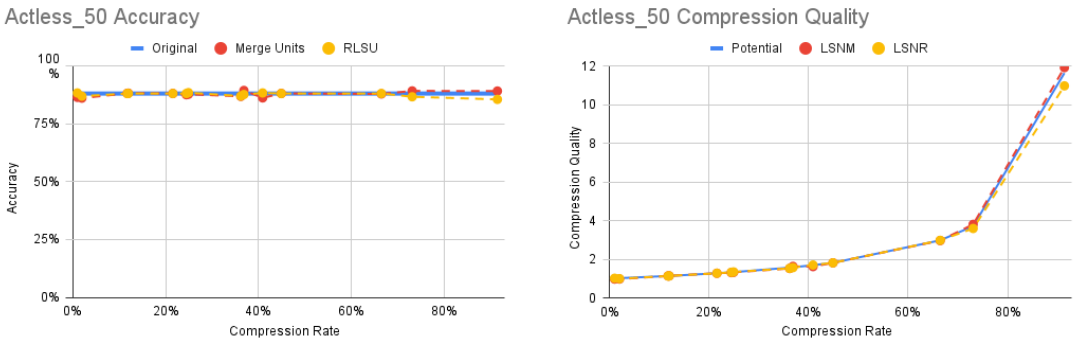


Figure 7.16: Accuracy and CQ diagram for the model Actless_50

In the diagrams in figure 7.8 can be seen, that both LSNR and LSNM perform equally excellent being almost equal to the *Potential*.

On average the Least Significant Node Remover performs better than Least Significant Node Merger, with few exceptions. The best overall results can be found at the “Actless” and “Full_Dense” models (in figures:7.15, 7.16, 7.11, 7.12), where the results are near the *Potential*. The Results tend to oscillate strongly suggesting that removing extra nodes can sometimes increase the quality of the results. This is further confirmed in 7.11 where the removing of nodes lead to going beyond the potential.

With Smart Training

To evaluate the influence of **Smart Train** on the results of both Least Significant Node Remover and Least Significant Node Merger, the models with the strongest deviation to the *Potential* were chosen. The models were then reshaped this time with **Smart Train**.

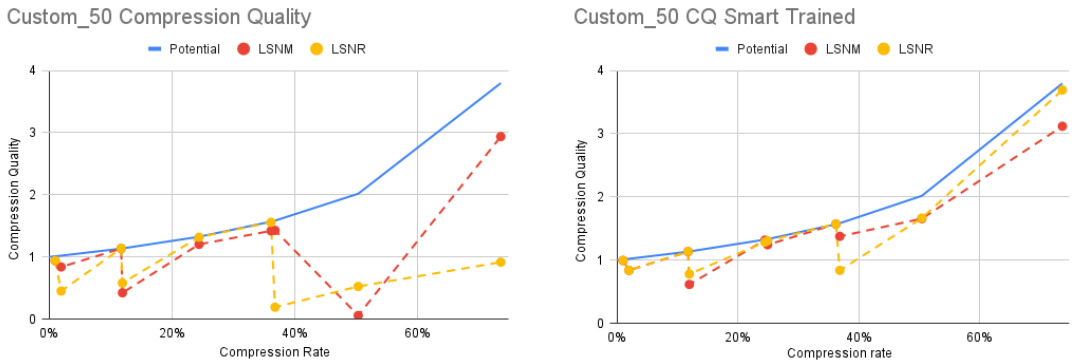


Figure 7.17: CQ diagram for the model reshaping of Custom_50 not and smart trained

The result of using **Smart Train** can be seen in figure 7.17 on the right side, in contrast to the left diagram(without **Smart Train**) the CQ is much closer to the *Potential* resulting in an overall better performance. And values close to zero were significantly improved by the inclusion of **Smart Train** in the reshaping process.

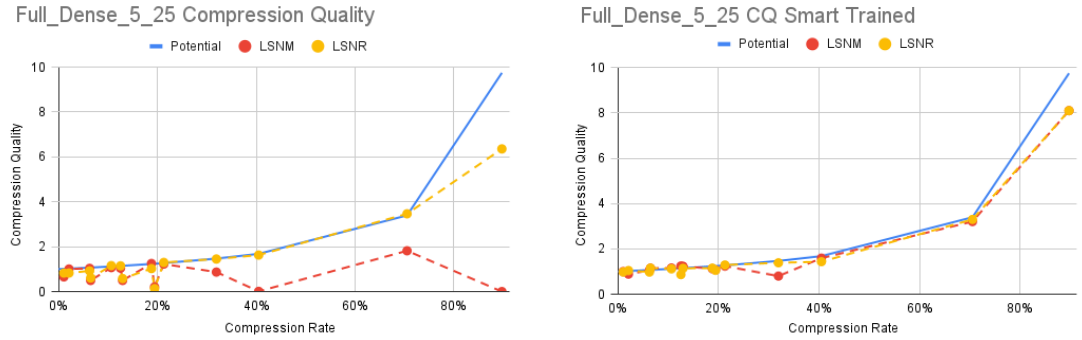
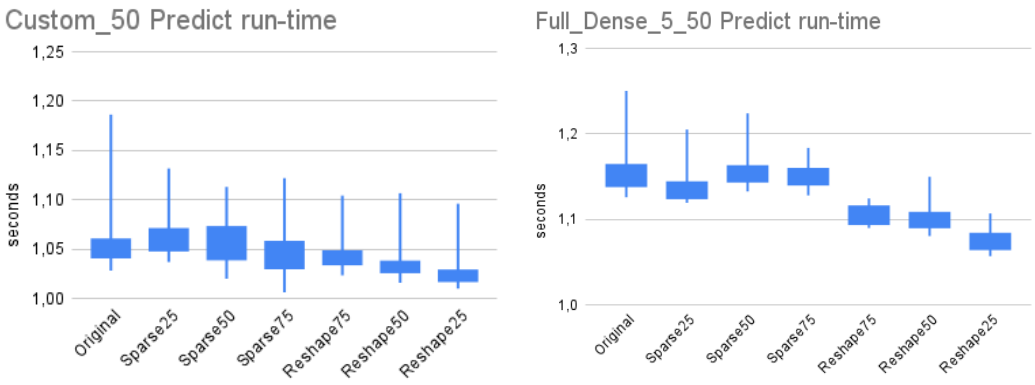


Figure 7.18: CQ diagram for the model reshaping of Full_5_Dense_50 not and smart trained

The Influence of **Smart Train** is even greater in this example(fig. 7.18), models unusable in the normal reshaping were significantly improved by the **Smart Train** process. The values reaching zero in the normal variant are now nearly equal to the *Potential*.

7.4 Runtime Evaluation

To evaluate the influence of compression on the runtime, the two models Custom_50 and Full_Dense_5_50 were first compressed with pruning methods to create three pruned models with 25, 50 and 75 percent sparsity and three reshaped models where Least Significant Node Merger was used to reduce the size of each layer by 25, 50 and 75 percent. Each model and the compressed variants were then used to predict 100.000 randomized entries 30 times to gain the data displayed in figure 7.19.



(a) Prediction run-time of models originating from Custom_50

(b) Prediction run-time of models originating from Full_Dense_5_50

Figure 7.19: Candlestick chart of prediction time

The candlestick chart displays the 30 time values of each model the thin lines being the longest and shortest times and the thick part being the values ranking 10 to 21. The diagrams visualize that no significant change of runtime is achieved by pruning layers, but the reshaping process can achieve smaller prediction time. This reflects the fact that pruned weights are still calculated during inference and do not result in better run-times.

8 Conclusion and Outlook

The two goals of this thesis were the design and implementation of compression methods and the design and implementation of the "ShowDiff" application, which is able to visualize Neural Network (NN)s to compare the original NN against the compressed one and the execution of the implemented compression methods inside the application.
Two new compression methods were implemented. The first being a pruning method with the name of Least Significant Path Pruner (LSPP), which calculates the shortest path from each output node of the NN to the input nodes and prunes the weights of edges commonly found in the shortest paths. Secondly, the Least Significant Node Merger (LSNM) method as a reshaping method, which merges the nodes between two layers by merging the parts of the weight matrices describing the layers effectively reducing the number of nodes in that layer. On top of these two different methods were implemented such as the existing methods of weight pruning and weight factorization and straight forward variants such as simple pruning, Least Significant Node Remover and Redundant Node Remover.

Beside the goals a procedure was implemented with the name of **Smart Train** , to improve the results of reshaping methods. By generating values to train parts of the compressed NN with the corresponding parts of the original NN.

The "ShowDiff" application was implemented which enables an user to import a NN and execute compression methods on it. The resulting NNs can then be visually compared against each other or the original model. The visualization displays NNs as a graph and can highlight changes from compression methods such as pruned edges, removed nodes and added layers.

Finally, the implemented experimental compression methods ware compared against the simpler variants to determine the effectiveness of the experimental methods. When it comes to Least Significant Path Pruner (LSPP) the method performs better than the simple pruner in low sparsity and equally good in higher sparsity. A shortcoming of LSPP is that large NNs result in low sparsity or long run-times. The Least Significant Node Remover (LSNR) performs in average better than the experimental Least Significant Node Merger (LSNM) method. But it occurs that the LSNM has better results than LSNR.

In future work, the purposed methods could be extended as follows. The application functions as a framework to implement different compression methods and test them against each other, therefore the expansion of the application is possible by implementing additional compression models.
In the current state **Smart Train** is only featured in reshaping methods, because the training could overwrite the pruned weights reversing the pruning in the process. The idea behind **Smart Train** could be expanded to the pruning methods by designing and implementing custom Keras layers wrapping the pruned layers to hinder changes on already pruned weights.

Bibliography

[BL16] S. Bhattacharya, N. D. Lane. "Sparsification and Separation of Deep Learning Layers for Constrained Resource Inference on Wearables". In: New York, NY, USA: Association for Computing Machinery, 2016. ISBN: 9781450342636. DOI: 10.1145/2994551.2994564. URL: https://doi.org/10.1145/2994551.2994564 (cit. on pp. 13, 15).

[Bokeh] *Bokeh*. URL: https://bokeh.org/ (cit. on p. 15).

[FC18] J. Frankle, M. Carbin. "The Lottery Ticket Hypothesis: Training Pruned Neural Networks". In: *CoRR* abs/1803.03635 (2018). arXiv: 1803.03635. URL: http://arxiv.org/abs/1803.03635 (cit. on pp. 14, 17).

[GRAPHVIZ] *Graphviz - Graph Visualization Tools*. URL: https://gitlab.com/graphviz/graphviz (cit. on p. 15).

[HPTD15] S. Han, J. Pool, J. Tran, W. J. Dally. "Learning both Weights and Connections for Efficient Neural Networks". In: *CoRR* abs/1506.02626 (2015). arXiv: 1506.02626. URL: http://arxiv.org/abs/1506.02626 (cit. on p. 30).

[Keras] *Keras*. URL: https://keras.io (cit. on p. 14).

[KerasPruner] *Pruning comprehensive guide*. URL: https://www.tensorflow.org/model_optimization/guide/pruning/comprehensive_guide (cit. on p. 30).

[KerasVIZ] *A Python Library for Visualizing Keras Models*. URL: https://github.com/lordmahyar/keras-visualizer (cit. on p. 15).

[LBG+16] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, F. Kawsar. "DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices". In: *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. 2016, pp. 1–12. DOI: 10.1109/IPSN.2016.7460664 (cit. on p. 15).

[LBM+16] N. D. Lane, S. Bhattacharya, A. Mathur, C. Forlivesi, F. Kawsar. "DXTK: Enabling Resource-efficient Deep Learning on Mobile and Embedded Devices with the DeepX Toolkit". In: *Proceedings of the 8th EAI International Conference on Mobile Computing, Applications and Services, MobiCASE 2016, Cambridge, UK, November 30 - December 01, 2016*. Ed. by F. Kawsar, P. Zhang, M. Musolesi. ACM, 2016, pp. 98–107. DOI: 10.4108/eai.30-11-2016.2267463. URL: https://doi.org/10.4108/eai.30-11-2016.2267463 (cit. on pp. 15, 30).

[LS18] H. Lee, J. Shin. "Anytime Neural Prediction via Slicing Networks Vertically". In: *CoRR* abs/1807.02609 (2018). arXiv: 1807.02609. URL: http://arxiv.org/abs/1807.02609 (cit. on p. 15).

[LWF+15] B. Liu, M. Wang, H. Foroosh, M. Tappen, M. Penksy. “Sparse Convolutional Neural Networks”. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 806–814. DOI: 10.1109/CVPR.2015.7298681 (cit. on p. 14).

[Panel] *Panel*. URL: https://panel.holoviz.org/ (cit. on p. 15).

[Performance per Watt] *Performance per Watt Is the New Moore’s Law*. URL: https://www.arm.com/blogs/blueprint/performance-per-watt (cit. on p. 11).

[XLG13] J. Xue, J. Li, Y. Gong. “Restructuring of deep neural network acoustic models with singular value decomposition.” In: *Interspeech*. 2013, pp. 2365–2369 (cit. on p. 29).

[Yen71] J. Y. Yen. “Finding the K Shortest Loopless Paths in a Network”. In: *Management Science* 17 (1971), pp. 712–716 (cit. on p. 24).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature