



UNIVERSITÄT  
KOBLENZ · LANDAU

Fachbereich 4: Informatik

# Visualization of Neural Networks

## Masterarbeit

zur Erlangung des Grades Master of Science (M.Sc.)  
im Studiengang Computervisualistik

vorgelegt von  
Julian Rogawski

Erstgutachter: Prof. Dr.-Ing. Stefan Müller  
(Institut für Computervisualistik, AG Computergraphik)  
Zweitgutachter: Bastian Kraymer M.Sc.  
(Institut für Computervisualistik, AG Computergraphik)

Koblenz, im Juli 2020

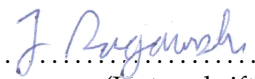
## Erklärung

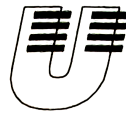
Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ja    Nein

Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.       

Großholbach, 09.07.2020  
(Ort, Datum)

  
(Unterschrift)



## Aufgabenstellung für die Masterarbeit

Julian Rogawski

(Mat. Nr. 212 100 730)

### Thema: Visualisierung neuronaler Netzwerke

Neuronale Netzwerke sind eine beliebte Deep Learning Methode im Bereich der künstlichen Intelligenz. Diese kommen oft für aktuelle Probleme im Bereich der Computerlinguistik und dem Maschinellen Sehen zum Einsatz und erzielen bei vielen Problemen state-of-the-art Ergebnisse. Oftmals stellen die neuronalen Netzwerke eine Art Blackbox dar und sind für den Menschen schlecht einsehbar. Eine Herausforderung ist es, eine Visualisierungsmethode zu finden, die ein gutes Maß an Nutzbarkeit und Ästhetik kombiniert.

Ziel dieser Arbeit ist es, eine Visualisierung einfacher Netzwerkarchitekturen zu konzipieren, welche den Einfluss von einzelnen Teilen der Architektur repräsentiert. Diese soll mit bekannten Visualisierungen und Analysemethoden verglichen und auf Nutzbarkeit zur Auswahl effizienter Netzwerkarchitekturen evaluiert werden.

Die inhaltlichen Schwerpunkte der Arbeit sind:

1. Recherche und Einarbeitung im Bezug auf existierende Visualisierungsmethoden
2. Konzeption der Visualisierung
3. Implementierung
4. Bewertung & Dokumentation der Ergebnisse

Koblenz, den 11.12.2019

- Prof. Dr. Stefan Müller -

- Julian Rogawski -

## **Zusammenfassung**

Künstliche neuronale Netze sind ein beliebtes Forschungsgebiet der künstlichen Intelligenz. Die zunehmende Größe und Komplexität der riesigen Modelle bringt gewisse Probleme mit sich. Die mangelnde Transparenz der inneren Abläufe eines neuronalen Netzes macht es schwierig, effiziente Architekturen für verschiedene Aufgaben auszuwählen. Es erweist sich als herausfordernd, diese Probleme zu lösen. Mit einem Mangel an aufschlussreichen Darstellungen neuronaler Netze verfestigt sich dieser Zustand. Vor dem Hintergrund dieser Schwierigkeiten wird eine neuartige Visualisierungstechnik in 3D vorgestellt. Eigenschaften für trainierte neuronale Netze werden unter Verwendung etablierter Methoden aus dem Bereich der Optimierung neuronaler Netze berechnet. Die Batch-Normalisierung wird mit Fine-tuning und Feature Extraction verwendet, um den Einfluss der Bestandteile eines neuronalen Netzes abzuschätzen. Eine Kombination dieser Einflussgrößen mit verschiedenen Methoden wie Edge-bundling, Raytracing, 3D-Impostor und einer speziellen Transparenztechnik führt zu einem 3D-Modell, das ein neuronales Netz darstellt. Die Validität der ermittelten Einflusswerte wird demonstriert und das Potential der entwickelten Visualisierung untersucht.



## **Abstract**

Artificial neural networks is a popular field of research in artificial intelligence. The increasing size and complexity of huge models entail certain problems. The lack of transparency of the inner workings of a neural network makes it difficult to choose efficient architectures for different tasks. It proves to be challenging to solve these problems, and with a lack of insightful representations of neural networks, this state of affairs becomes entrenched. With these difficulties in mind a novel 3D visualization technique is introduced. Attributes for trained neural networks are estimated by utilizing established methods from the area of neural network optimization. Batch normalization is used with fine-tuning and feature extraction to estimate the importance of different parts of the neural network. A combination of the importance values with various methods like edge bundling, ray tracing, 3D impostor and a special transparency technique results in a 3D model representing a neural network. The validity of the extracted importance estimations is demonstrated and the potential of the developed visualization is explored.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.1.1	Model Size . . . . .	1
1.1.2	Black Box Problem . . . . .	2
<b>2</b>	<b>Fundamentals</b>	<b>2</b>
2.1	Neural Networks . . . . .	2
2.1.1	Fine-tuning & Feature Extraction . . . . .	4
2.1.2	Regularization techniques . . . . .	5
2.1.3	MNIST . . . . .	6
2.1.4	Batch Normalization . . . . .	7
2.2	Kernel Density Estimation . . . . .	8
2.3	Impostor . . . . .	8
<b>3</b>	<b>Related Works</b>	<b>10</b>
3.1	Neural Network Representation . . . . .	10
3.2	Neural Network Optimization . . . . .	12
3.2.1	Pruning . . . . .	14
3.2.2	Network Slimming . . . . .	15
3.3	Graph Processing . . . . .	16
3.3.1	Edge Bundling . . . . .	16
3.3.2	Kernel Density Estimation-based Edge Bundling . . . . .	16
3.3.3	3D Edge Bundling . . . . .	18
3.4	Rendering Techniques . . . . .	18
3.4.1	Cube Impostor . . . . .	18
3.4.2	Transparency . . . . .	19
<b>4</b>	<b>Concept</b>	<b>19</b>
4.1	Importance . . . . .	20
4.2	Node & Edge Bundling in 3D . . . . .	21
4.3	Rendering . . . . .	23
4.3.1	Cuboid Impostor for Ellipsoids . . . . .	24
4.3.2	Importance dependent Transparency . . . . .	24
<b>5</b>	<b>Implementation</b>	<b>25</b>
5.1	Importance calculation . . . . .	26
5.2	Neural Network Processing . . . . .	27
5.2.1	Grid . . . . .	27
5.2.2	Node Clustering . . . . .	27
5.2.3	Edge Bundling . . . . .	29
5.3	Neural Network Rendering . . . . .	32

<b>6</b>	<b>Analysis and Evaluation</b>	<b>32</b>
6.1	Importance . . . . .	35
6.1.1	Overall Importance . . . . .	36
6.1.2	Class Importance . . . . .	39
6.2	Bundling Comparison . . . . .	42
<b>7</b>	<b>Conclusion</b>	<b>44</b>
7.1	Usability . . . . .	44
7.2	Future work . . . . .	45

# 1 Introduction

## 1.1 Motivation

Neural networks are not a new technology, but have been gaining renewed interest with improving hardware and computational power and became a large field of research. With the recent developments regarding the growing complexity and size of neural networks, new problems are emerging. Two key factors are the growing size and the incomprehensible inner workings of complex networks.

### 1.1.1 Model Size

With performance gains particularly in natural language processing and computer vision, neural networks are being used for an expanding range of tasks with increasingly large models. The technological advances in GPU and specialized neural processing units open the door for models with billions of parameter [4][23].

For image classification challenges the winning models growing from architectures using 8 to more than 100 layers [23]. With ResNet for example having 152 layer, 60 million parameters and requiring 20 giga floating point operations for the use on a single  $224 \times 224$  image [23]. The trend for these tasks still seems to lead to a growth in model size, as indicated by recent developments [27].

But with these huge neural networks problems are emerging. The increasingly costly computations and memory consumption are not always readily available and the training of these neural networks is expensive in hardware and/or computation time. On smaller devices even the application becomes impossible due to the size alone. Following [24] there are three major constraints of neural networks:

- memory footprint
- computational cost
- power consumption

Coupled with the fact that “Over-parameterization is a widely recognized property of deep neural networks” [24], these problems lead to research that focuses on effectively reducing model sizes while maintaining performance. The problem with creating more efficient models can be tackled by identifying and reducing unnecessary parameters.

### 1.1.2 Black Box Problem

Advanced AI has the disadvantage that it becomes too complex to completely comprehend, which limits the understanding of the inner workings. The lack of transparency in Deep Learning methods makes it hard to analyze. Neural networks in particular are difficult to design and the main method for creating high-performance models is the use of empirically proven architectures, that are used for similar tasks. The black box problematic is also described as the main reason of compression approaches are not seeing wide adoption for reducing model sizes [4].

The black box problem seems to be a major obstacle for more intricate visualization techniques as well. The most common visualizations regarding architecture often contain information that describe the structure of a network but ignore trained parameters. The sheer quantity of these parameters needs filtering or abstraction to be digestible. Some research is done regarding the processing of inputs throughout the network, but these do not directly address the architecture of the model and are often still too complex to follow. The lack of existing 3D representations is therefore explored and an own novel method will be introduced in this thesis.

## 2 Fundamentals

This section provides a brief overview of basic knowledge on the topics covered in this thesis. First the basics of neural networks are explained, followed by some more specific topics in this area of research. The next part introduces a commonly used statistics technique Kernel Density Estimation, which plays a key part in some more intricate methods explored in this thesis. Lastly a rendering technique is presented, which is adopted for visualizing neural networks.

### 2.1 Neural Networks

Neural networks play a big part in machine learning and deep learning. A neural network can be described as a graph consisting of linked neurons. A neuron can be described as a function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  from [15]:

$$f(\vec{x}) = a(w_1x_1 + \dots + w_nx_n + b) \quad (1)$$

Where  $a$  is the activation function  $a : \mathbb{R} \rightarrow \mathbb{R}$ . There is a variety of functions, that can be chosen. From a simple linear function to a more complex function like the sigmoid function. The choice is often influenced by empirical

data.  $\vec{w}$  are the weights for incoming data to the neuron, which are multiplied component wise with the input vector  $\vec{x}$ . Every input dimension is associated with a weight and can be seen as an incoming edge.  $b$  is the bias and is added to the combination of weights and inputs and fed together into the activation function. A visualization of the equation is presented in 1.

By combining multiple neurons we get a simple neural network as pre-

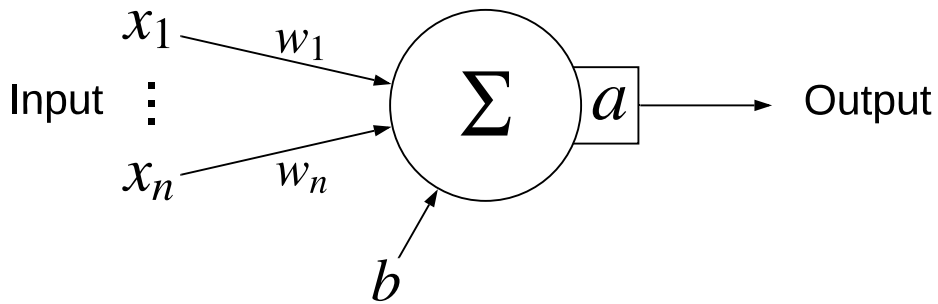


Figure 1: A single neuron

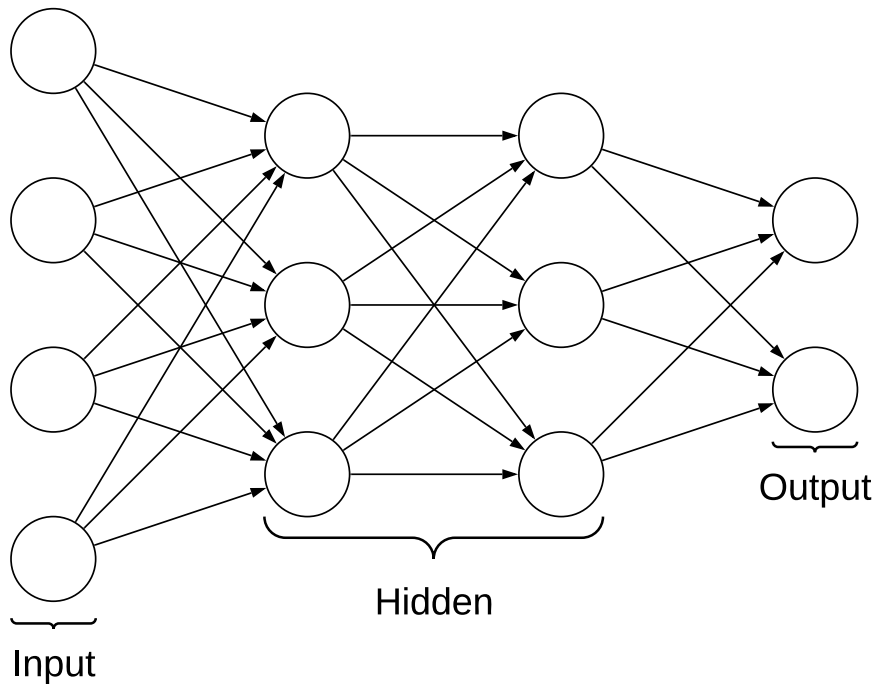


Figure 2: A simple fully-connected neural network

sented in 2. A network is generally divided into layers consisting of neurons. There is usually one input layer, where the data flows into a network and one output layer, which can also have a various number of nodes. As

an example a neural network model can use the pixel data of an image as its input with one node for every pixel. When using this network for recognizing animals shown in the picture, the output nodes can be one for every animal the network should recognize.

A neural network can be constructed with any number of hidden layers between input and output. These layers are often described as the black box part, because in most cases no direct interaction with these layers is required when using a neural network. These inner workings are especially difficult to comprehend with more complex networks.

Figure 2 shows a simple type of network only consisting of fully connected layers, which are the main focus of this thesis. Nodes of fully connected layer have one edge for each node of the following layer. Another type are convolutional neural networks (CNN) consisting of convolutional layers paired with fully connected ones. These are the most simple and common types of layer.

The parameters of a network are usually randomly initialized and are changed through backpropagation [15]. This process is referred to as training.

The final output can be seen as a prediction, which is compared to an expected value corresponding to the input data. The measurement of how far the prediction deviates is called the loss function, which can be chosen, as desired similar to the activation function in (1). For example the well known calculation of the mean squared error (MSE) can be used over multiple losses and in the backpropagation method. This loss is calculated through every layer from output to input in an effort of reducing this value. Different optimization methods can be paired with backpropagation to iteratively reduce the loss and in turn improve the predictions of a neural network, by changing the parameters (like the weights or bias mentioned in (1)). For the sake of simplicity the optimization methods are not further explained.

The training process uses a set of labelled data, which consists of input data samples paired with labels or expected/true output data for the corresponding input. [15] provides a more detailed description. This thesis focuses on neural networks used for classification tasks, which uses data labelled at least as one of multiple potential classes. Through training the parameter, the neural network learns to predict the labels for given data. When a neural network finished training, it can be used to predict these labels for any suitable data, which is called inference.

### **2.1.1 Fine-tuning & Feature Extraction**

Fine-tuning for neural networks is a valid technique for updating already trained models [14]. This technique can be seen as a continuation of the

learning process of a neural network model.

Usually a pre-trained model is restructured to fit a new task in fine-tuning, for instance by replacing the last layer with a new one matching similar but new classification. The parameters of the modified network are then optimized on a set of data for the new task. With this method it is possible to train multiple networks each with their own specialization as described in [22].

This can be highly beneficial when using large data sets and long training times. By using more general data, a model can be trained and general features can be learned. Afterwards the same model can be used repetitively for a new task focusing on different and more specialized data, without re-training a completely new model on the initial large data. The specialized data is usually much smaller and the modified models are therefore trained much faster.

A prominent example in natural language processing is the pre-trained model BERT [7] freely available for any language processing task. This model is amongst other data pre-trained on 2,500 million words as contiguous sequences of sentences from English Wikipedia. The base version of BERT has 110 million parameters and the large version 340 million. The complete training from scratch on such a large model is very resource intensive, therefore using a pre-trained model and fine-tuning it with much less data is very convenient.

In feature extraction a pre-trained model similar to fine-tuning is used. The outputs of one or more layers are used as inputs for new layers. In this case while training a new model, the parameter of the old layers stay fixed [22]. The identification of complex features learned in the pre-trained model remain preserved and the newly added layers learn to utilize it on the new task.

These techniques are also used in combination and are compromising between fine-tuning and feature extraction [22]. A combination of both is applied for this thesis and explained later in 4.1.

### **2.1.2 Regularization techniques**

Sparsity regularizations have been explored in an effort to combat the computational costs and memory intensity of neural networks [29]. The focus for this work lies on regularization enforced by simply changing the loss function during training. These methods are easy to implement and do not need changes in the architecture of a model.



$L_1$  is called Lasso Regression and adds the absolute value of the parameter as a penalty to the loss function. The following equation is taken from [25]:

$$L_1 = \lambda \sum_{j=1}^p |\beta_j| \quad (2)$$

$\beta$  is the parameter, which should be regularized and  $\lambda$  defines the scale at which this term should be applied. This term lets the parameters tend towards 0, if they don't have a great impact while training and is therefore useful for feature extraction [25].

As suggested by [4]  $L_1$  is often used in optimization because it leads to a wanted sparsity.

$L_2$  is called Ridge Regression and works similar to the Lasso Regression. The following equation is taken from [25]:

$$L_2 = \lambda \sum_{j=1}^p w_j^2 \quad (3)$$

Instead of using the absolute value of a parameter, the squared value is added to the loss function, resulting in a heavy penalty for large values. This is less desirable for feature extraction and leads to an increased generalization.

$L_{1,2}$  is the combination of both and adds simply both terms to the loss during training.

### 2.1.3 MNIST

MNIST [19] is a openly available database for handwritten digit images and is an established standard in testing machine learning algorithms [6]. Some sample images are presented in 3.

An image consist of  $28 \times 28 = 784$  grayscale pixels where each is labelled as one of the 10 possible digits and the whole database contains 70k labelled images. These images are split into 60k training samples and 10k test samples.

Using this already revised data saves time that is otherwise needed for preparing real world data. Results for machine learning methods applied to MNIST can conveniently be compared with many other methods already providing evaluation data for this exact task. Especially neural network

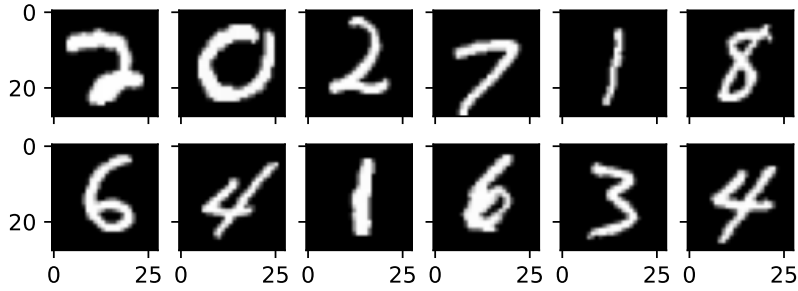


Figure 3: MNIST example images

classifier, which are used for this thesis, are prominently applied on MNIST and achieve high performances [6].

#### 2.1.4 Batch Normalization

Batch Normalization [13] is a method used to normalize layer inputs of neural networks and is compatible with any type of activation function in a network. Special batch normalization layers can easily be added between layers of an existing model and trained through backpropagation without any additional overhead to the models training process. The distribution of layer inputs usually changes during the training, which leads to a constant readjustment of the following layers to the dynamic distributions. At this point, batch normalization intervenes to improve the stability of these distributions, proven to enhance the overall training process.

A batch normalization Layer uses batches of data to compute mean and variance of the activation values for each batch during training. Using batches of data is already common in training neural network models, therefore adding no additional overhead to the regular training process. The average of the mean and variance of all batches is used as estimates for the overall mean and variance assumed for all data, which is used in inference. The equation for calculating the output values of a batch normalization layer is defined as following:

$$y = \gamma \cdot \hat{x} + \beta \quad (4)$$

The parameter  $\gamma$  and  $\beta$  are also learned through backpropagation.  $\gamma$  scales the modified input and determines the strength of the inputs influence in subsequent layers.  $\hat{x}$  is the normalized input and is calculated with estimates for  $E[x]$  and  $Var[x]$  and  $\epsilon$  is a small value added for numerical stability:

$$\hat{x} = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} \quad (5)$$

This method effectively speeds up the training process and stability. In fact it also serves as a regularizer (2.1.2) for the model and in some cases improves the performance [13]. Batch normalization layers are also commonly used after convolution layers to increase performance [23]. The  $\gamma$  value from (4) is also proven to be useful for identifying unimportant parts of a neural network model [23]. The technique of processing the  $\gamma$  value to extract an importance measure is applied in this thesis to filter and visualize important parts of a network.

## 2.2 Kernel Density Estimation

Kernel Density Estimation (KDE) is “the most well-known approaches to estimate the underlying probability density function of a dataset” [3]. The estimator requires minimum input and is very flexible to apply to various types of data.

KDE is basically smoothing single data points in small bumps ranging over a certain area depending on the bandwidth  $h$ . The accumulation of these stretched bumps can be seen as the probability of samples being at certain positions over a range of possible values like coordinates in a 3D space. With this method it is possible to create an estimate of the real probability function by only using a number of samples.

The estimator is described as the following equation from [12]:

$$\rho(x) = \sum_{i=1}^N \int_{y \in e_i} K\left(\frac{x-y}{h}\right) \quad (6)$$

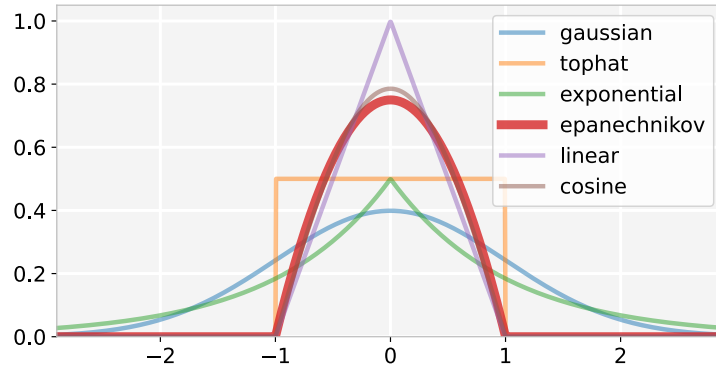
$\rho(x) : \mathbb{R}^2 \rightarrow \mathbb{R}^+$  is the estimated probability density function using the density kernel  $K : \mathbb{R}^2 \rightarrow \mathbb{R}^+$ .

There are many choices for the types of functions that can be used for  $K$ . An overview of the shapes for some popular choices is depicted in figure 4. For this thesis the highlighted *Epanechnikov* plays a major part and is defined as  $K(x) = 1 - \|x\|^2$ . This function “optimally approximate the density map in a minimal variance sense” [12].

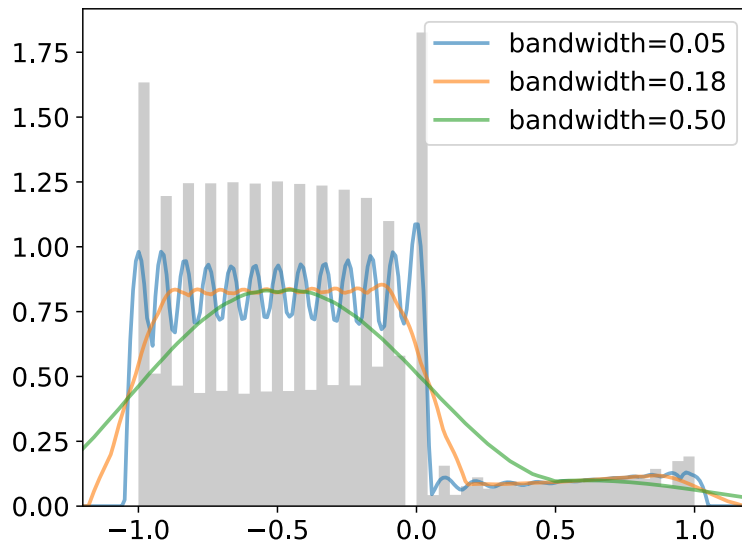
The effect of bandwidth  $h$  is illustrated in figure 5. The choice of  $h$  needs to be considered carefully and heavily depends on the used data.

## 2.3 Impostor

Impostors are used in many graphics applications dealing with large amounts of objects that need to be rendered. One example is the field of molecular visualization [28]. Instead of using complex meshes for rendering, textured 2D-billboards are used to improve the performance.



**Figure 4:** Different kernel functions with *Epanechnikov* highlighted



**Figure 5:** A histogram from data points and density estimations with varying bandwidths

A ray-sphere intersection computation [9] can be used for a precise representation of a sphere, which makes it attractive to represent the many atoms of molecules as spheres in real-time applications, where large organic molecules consist of more than 50k atoms [28].

The following equation from [9] needs to be solved to get the intersection positions of a ray and a sphere:

$$\underbrace{(\vec{d} \cdot \vec{d})}_a t^2 + 2 \underbrace{(\vec{f} \cdot \vec{d})}_b t + \underbrace{\vec{f} \cdot \vec{f} - r^2}_c = at^2 + bt + c = 0 \quad (7)$$

The solution for this quadratic equation from [9] is:

$$t_{0,1} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (8)$$

$\vec{d}$  is the ray direction and  $\vec{f} = \vec{o} - \vec{G}$  and  $\vec{G}$  is the center position of the sphere.

The discriminant  $b^2 - 4ac$  falls below 0 there are no intersections and the fragment of an impostor can be discarded. Otherwise  $t_{0,1}$  can be used to calculate both intersection points.

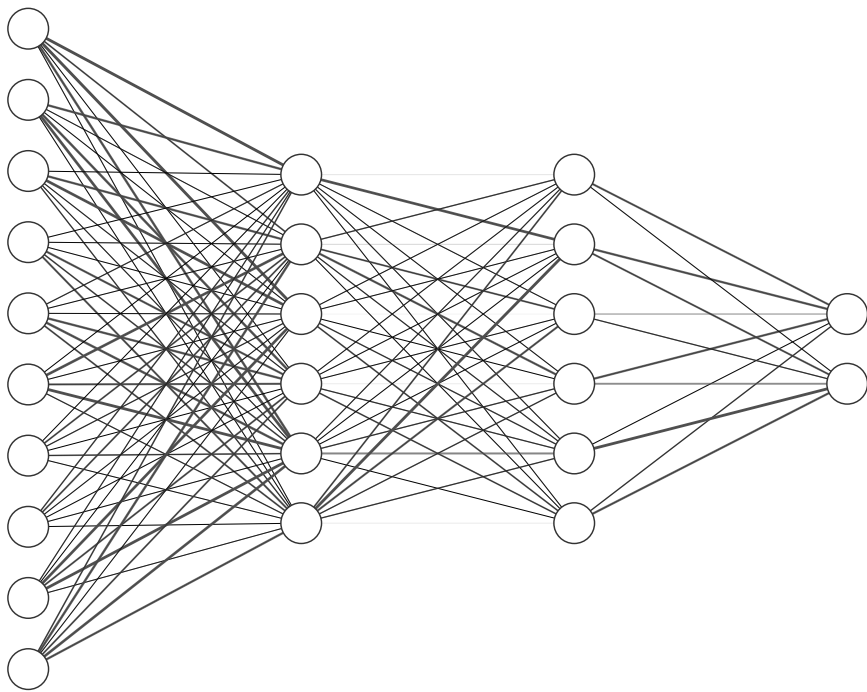
Since networks are huge graphs with nodes and edges, the idea of using impostors for their representation is reasonable. The nodes and edges can be interpreted as two simple primitives that are rendered repeatedly at different positions. The use of sphere impostor for nodes and a modified version for edges is explored in this thesis.

## 3 Related Works

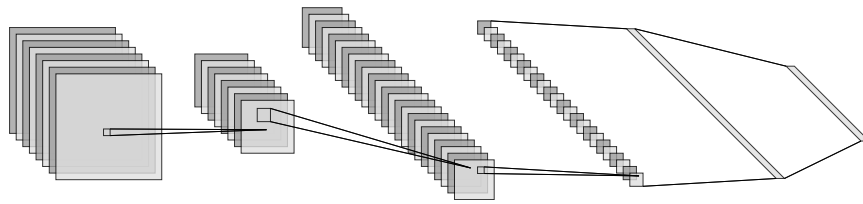
Methods from multiple areas of research are combined in this work. This section is split into four parts each focusing on one area. In order to reflect the current state of visualization of neural networks, several methods are presented in 3.1. Section 3.2 centers around optimization of neural networks, providing some interesting methods to identify the important parts of a neural networks architecture. Chapter 3.3.1 reviews techniques for decluttering huge graphs for visualization and 3.4 provides insight in relevant rendering technologies.

### 3.1 Neural Network Representation

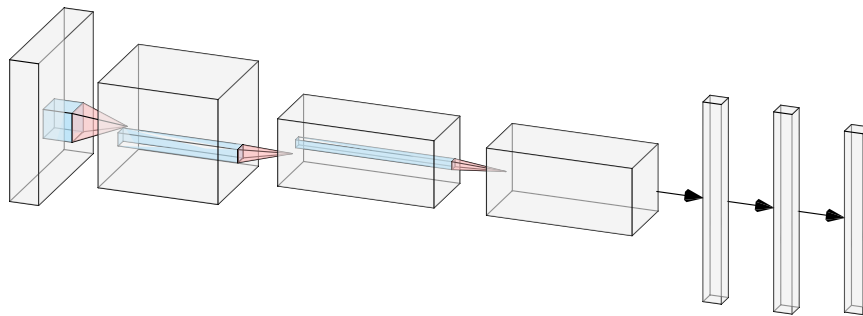
A classic illustration used to represent small fully-connected neural networks is shown in figure 6. These types of node-link diagrams are often used in an educational context, but bring hardly any benefit for medium to



**Figure 6:** Fully-connected neural network represented with nodes and edges. The edges vary in size and opacity depending on their weight. Created using [20]



**Figure 7:** Illustration of a convolutional neural network. Created using [20]



**Figure 8:** Illustration of a convolutional neural network. Created using [20]

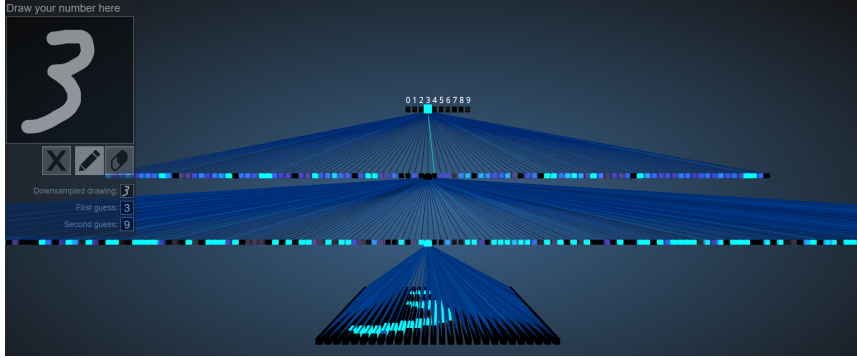
large models [10]. Stronger edge weights are highlighted in 6, but except for small scale models, these tend to get too crowded to gain any insight about the model.

For convolutional neural networks, which use convolutional layer, two prevalent illustration types emerged. Figure 7 shows the “LeNet style” [20] first established in [18] and figure 8 shows the “AlexNet style” [20] first established in [16]. Both represent the convolutional layer with special symbols and are more abstract without showing any detail about the trained parameters. These illustrations usually add labels to describe details about the different sizes of layers.

An example of existing 3D visualization is the interactive tool, that is introduced in [10] focusing on neural network models trained on MNIST (see 2.1.3) data. This tool combats the overwhelming amount of edges in a fully-connected layer by letting the user chose only small subsets of edges at a time. As shown in figure 9 the user can write a digit and the tool highlights the nodes with high activation values in the visualization.

### 3.2 Neural Network Optimization

Neural Network Optimization is a current topic regarding the computationally expensive and memory intensive large neural networks, where there is still difficulty finding usage in situations with insufficient hardware



**Figure 9:** An interactive tool visualizing the node activations from a drawn input and showing incoming edge weights while hovering over the nodes. [10]

**Table 1:** Summarization of different approaches for model compression and acceleration taken from [4]

Theme Name	Description	Applications
Parameter pruning and sharing	Reducing redundant parameters which are not sensitive to the performance	convolutional layer and fully connected layer
Low-rank factorization	Using matrix/tensor decomposition to estimate the informative parameters	convolutional layer and fully connected layer
Transferred/compact convolutional filters	Designing special structural convolutional filters to save parameters	convolutional layer only
Knowledge distillation	Training a compact neural network with distilled knowledge of a large model	convolutional layer and fully connected layer



availability [23]. A characteristic of neural networks is the overparameterization, which suggests a high degree of potential optimization.

One way to optimize a neural network model is by compression and acceleration. In general, the goal is to reduce the required resources of neural networks while preserving model performance. Some methods show effectiveness in lowering both complexity and the overfitting problematic of neural networks [4]. A promising way to achieve this is by identifying and extracting insignificant and redundant parameters.

There are four main methods for compression and acceleration of neural networks, which are described in table 1

Pruning seems to be the most accessible method for a more general neural network visualization. It works either way for both convolution and fully-connected layer, that are traditionally the most common used types of layers. Pruning is also applicable on pre-trained models, providing a huge advantage in dealing with large neural networks by saving a lot of resources otherwise needed on training a model from scratch.

### 3.2.1 Pruning

Pruning is researched with the goal of taking on the problems described in 1.1.1 to help mitigate the substantial cost of inference of large neural networks. A reason for this is to increase the efficiency especially in environments with limited computational resources [24].

A special emphasis is set on preserving the accuracy of models after pruning. This entails the goal of cutting down non-informative parameters in the network. In Pruning methods unimportant parameters are identified and subsequently removed or disabled and it usually happens as a part of three sequential steps [24]:

- Training
- Pruning
- Fine-tuning

A drawback of the pruning method is the fine-tuning that is needed because of the sensitivity of neural networks towards pruning parameters. But the network performance critical aspects are negligible when serving a visualization method, that abstracts a model for general architecture analysis. Therefore the focus for this thesis is on the pruning step and on identifying the important parameters alone.

One of the more straightforward methods to achieve this, is by simply removing connections. Layer-wise magnitude-based pruning [21] expands

on it by defining a threshold for each layer and prune parameters not exceeding this threshold per layer.

A special emphasis is set on preserving the accuracy of models after pruning. This entails the goal of cutting down on non-informative parameter in the network. When pruning single parameter, special sparse layers are created, which is not always desired. Therefore many pruning methods focus on pruning whole nodes or even whole layers with their associated parameters. The difficulty in these types of pruning is the potential of changing the input of subsequent layers in unforeseen ways [4], which often result in great accuracy losses.

Another way of identifying unimportant parameters is by calculating the Hessian matrix of the loss function, which preserves a higher accuracy than related methods [4], but is more difficult to implement. There are similar concepts that take advantage of the commonly used  $L_1$  or  $L_2$  regularization (see 2.1.2), but need more iterations of backpropagation to achieve the necessary result.

It is therefore reasonable to assume, that this kind of method can be used for measuring the importance of parameters. This kind of measurement can then be used to highlight more significant parts of a neural network.

### 3.2.2 Network Slimming

Network slimming (a specific kind of pruning) is aiming at reducing the neural network model size, decreasing the run-time memory footprint and computing operations, while preserving the accuracy of the improved model [23]. The method identifies unimportant neurons and channels, which can be easily pruned from the model. For models that exceed a certain depth, even layers can be effectively removed. Although it is designed for convolutional layers, it can also be used for fully-connected layers and is therefore functional for a wide range of neural network models.

The method combines batch normalization with  $L_1$  regularization while training a neural network to filter channels or neurons not essential in the network by utilizing the scaling factors in batch normalization layers [23]. This entails the condition of training a neural network, before it can be effectively pruned. The uncompromising removal of channels and neurons can degrade the performance in unexpected ways, making it essential to fine-tune a pruned network to maintain the original accuracy values. Although [23] shows that the accuracy of pruned models sometimes exceeds the originals.

The advantage of Network Slimming over different methods becomes apparent by multiple factors:

- minimal impact on the training process of a neural network
- no need for specific software/hardware
- minimal programming overhead through existing neural network libraries

In [23] it is also suggested, that this method can be employed for architecture learning, prompting the speculation over its value for evaluating neural network architectures.

### 3.3 Graph Processing

At their core neural networks are dense directed graphs. Using visualization techniques that are common for small graphs are not useful for neural networks, because of the huge size of these networks. Fortunately there are methods that focus on this type of graphs. In this section some relevant work regarding the visualization of huge graphs are examined.

#### 3.3.1 Edge Bundling

An effective method for decluttering very dense graphs and therefore its usefulness in network analysis is given by edge bundling [12]. The edges of a processed graph are represented as “tightly bundled curves”[12], which improves the efficiency of visually presented graphs by highlighting specific parts of their structure. Generally the result is an improved readability of emerging cluster of nodes.

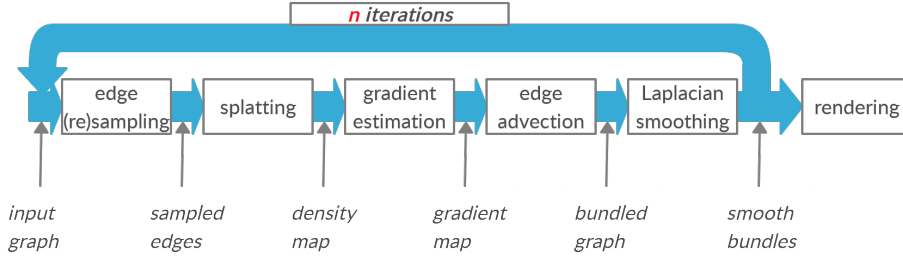
The general process of this method consists of the following parts and are usually implemented iteratively:

- construct a density map from the edges of the graph
- calculate the gradients for the edges using the density map
- advect the edges in gradient direction

#### 3.3.2 Kernel Density Estimation-based Edge Bundling

KDEEB [12] is one implementation example. It is entirely image-based and uses Kernel Density Estimation. This method meets the relevant requirements of being fast and simple to implement and it also bundles edges by applying KDE in an iterative fashion. In figure 10 adapted from [12] the process is summarized.

The vertices positions of a graph in a 2D plane are the input for this method and can be set by any kind of algorithm. The edges between the vertices



**Figure 10:** KDEEB pipeline adapted from [12]

are then sampled, which results in a representation of each edge as sets of points. The density map as the approximation of the real density function is calculated by using elliptical kernel. These kernels are applied between every two neighbouring samples of an edge, following the KDE technique described in 2.2.

The next equation describes the bundling operator  $B$  used in KDEEB and is based on the density estimation  $\rho$  described in (6):

$$\frac{dx}{dt} = \frac{h(t)\nabla\rho(t)}{\max(\|\nabla\rho(t)\|, \epsilon)} \quad (9)$$

The solution of this ordinary differential equation, which utilizes the gradient  $\nabla\rho$  of the density map defines the direction, in which the samples are moved during the advection phase. The samples of an edge are basically moved along the direction vector of the gradient, which is normalized and scaled by the bandwidth  $h$ .

Therefore the distance and advection speed is defined by the bandwidth  $h$ , which is the also the kernel bandwidth used in creating the density map  $\rho$ .  $h_{max}$  is the initial bandwidth and is set to the average inter-edge distance of the input graph [12].

$$h_i = \lambda^i h_{max} \quad (10)$$

Iterative bandwidth reduction reduces the advection speed, stabilizing the process and leading edges to converge to a local density maximum. Inaccuracies in density estimation and discretization errors resulting from edge sampling lead to artifacts that are displayed as zig-zag lines in bundles. To mitigate this problem, multiple Laplacian smoothing iterations of the edges are performed after each advection step. Well separated bundles of edges reveal a smooth graph structure after estimating densities & gradients, advecting samples and smoothing over multiple passes.

Different shading and blending techniques can be applied to render a final 2D image representing the graph and emphasizing the density of edge

bundles, resulting in increased information captured in the final visualization.

KDEEB seems promising in providing an easily adaptable method. The capability and possibilities of modifying the basic method is proven in the same work [12] by including user-specific obstacles, which are avoided by edges in the bundling process.

### **3.3.3 3D Edge Bundling**

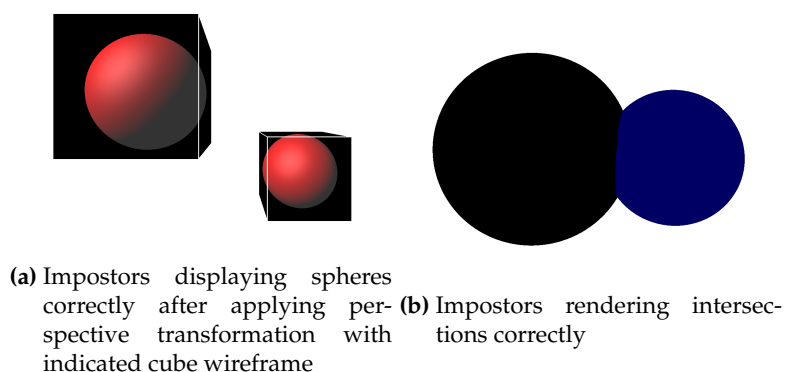
The application of edge bundling on 3D graphs is shown in [17]. Special grids are used instead of creating a large number of textures to cover the 3D space. To achieve higher resolution without taking up too much memory, grids with multiple varying sizes such as octrees or 3D Voronoi diagrams can be applied. [17] mentions a common occlusion problem occurring with 3D visualization and explores a technique to circumvent this issue by revolving the edge bundles around a globe that are applied on geographical data. This self-occlusion problem of 3D visualizations inspired the application of special rendering techniques in this thesis, which are described in the next chapter.

## **3.4 Rendering Techniques**

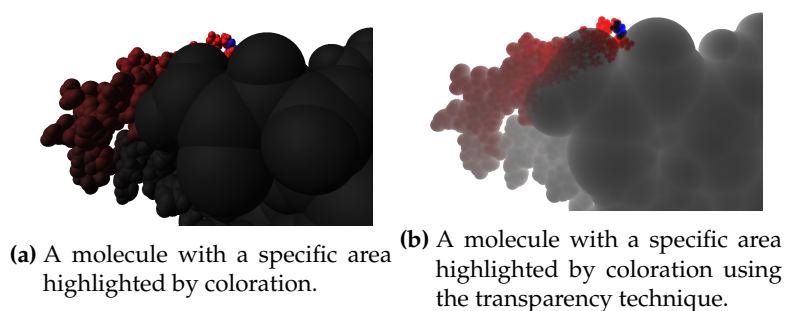
This section covers the two main techniques adopted for the visualization of neural networks introduced in this work. By Visualizing huge graphs in 3D, two major problems need to be addressed. On the one hand, the rendering of many nodes and edges in real time is not trivial, on the other hand the problem of self-occlusion arises especially in 3D. The following two chapters introduce two techniques explored in an effort to combat these problems in their combination.

### **3.4.1 Cube Impostor**

In [26] cube impostors in combination with the ray-sphere intersection calculations described in [9] are used to represent atoms of a molecule as spheres. Compared to using 2D-billboard as impostor, there is no need to displace the impostor in regards to the perspective transformation while rendering. These cube impostor are shown in 11. By using these impostors, rendering large amounts of similar object makes it more efficient and suitable for real-time applications.



**Figure 11:** Two Cube Impostor rendered displaying ray-traced spheres in different situations from [26]



**Figure 12:** Comparison of two differently rendered images of the same molecule from [26]

### 3.4.2 Transparency

As previously mentioned in [17] and [11] the self-occlusion of complex 3D-structures hides important internal information. In [26] a transparency technique is used to highlight specific parts of a molecular structure. The technique is shown in 12 and provides a promising way of emphasizing certain parts of a complex structure and is adopted in this thesis for highlighting important parts of a neural network graph.

## 4 Concept

A novel method is designed to visualize neural networks in an attempt to address the problems described in 1.1. The goal is to find a way to gain digestible insights about neural networks through processing and visualization with the focus on the architecture and the influence it has on the

performance. Several methods from different areas presented in 2 and 3 are modified and combined. This chapter presents the different modifications in theory and the hypothesized effects of these modifications. The combination of these methods results in a technique for processing an input neural networks and providing a 3D visualization.

## 4.1 Importance

Huge amounts of data such as the parameters of a neural network are difficult to visualize. When all parameters are displayed unfiltered, the presentation quickly becomes overwhelming, making it difficult to retrieve information from it. The most important factor in evaluating neural networks is how well they approximate specific outputs given the corresponding inputs. The accuracy of their predictions in comparison to the expected result is the main factor in choosing neural networks.

One way to declutter the neural network is to filter the parts of the neural network according to their importance for the accuracy of the whole neural network in order to highlight the relevant parts. This type of ranking parameters on different objectives like “absolute values, trained importance coefficients, or contributions to network activations or gradients” is quite common [2].

The importance of nodes and edges in a neural network can be estimated by simply using weights of edges between neurons. Unfortunately, for the weights in a graph, it cannot be assumed that the value immediately represents its importance for the graph’s structure.[30]. These weights therefore might not correlate with the real importance of edges in the sense of preserving the structure of graphs. Nodes combined with their edges might unexpectedly change the input of the following layer when changed [4]. Therefore simply using the parameters of a network is not sufficient for visualizing important parts of a neural network. To mitigate this problem, the method described in 3.2.2 is adopted. But instead of calculating one overall importance value, an importance vector is constructed for every node with one dimension regarding every output node  $i \in N$ :

$$I = (\iota_1, \dots, \iota_n) \quad \text{with} \quad \iota_i \in \mathbb{R}^+ \quad (11)$$

This is achieved by modifying an existing neural network model and training it on a tailored subset of data corresponding to the respective output node. Between every existing layer a batch normalization layer is added. These new layers normalize the outputs of the previous nodes and scaling them for the next layer. In the training process the scaling factors are optimized for each class and are interpreted as importance of their con-

nected input and output. The importance vectors for nodes and edges are extracted from the trained modified model, assuming the length of  $I$  as the overall importance for the network.

In principle, this is a combination of fine-tuning and feature extraction 2.1.1, where the model that will be visualized plays the role of the pre-trained model. The modifications are added for the new tasks the network will be trained on. In order to keep the original structure intact, all original layers are frozen and only the modifications in form of added layers are fine-tuned. The process is further explained in 5.1.

This importance vector is used as an attribute in processing a neural network and creating a 3D representation, describing their impact on different output nodes at the end of the network architecture.

## 4.2 Node & Edge Bundling in 3D

Edge Bundling is a way to declutter graphs by grouping edges into bundles and creating visible distinctions between them. Since neural networks are basically huge graphs, this method is applicable and offers a potential benefit for a graphical representation.

This technique is especially used for geographic data, which emphasizes the key role of node positions in a graph for Edge Bundling. Before it can be applied properly, the node positions must be determined. The main factor for the positions of the nodes is their layer in the neural network. Therefore the nodes of the different layers should be clearly separated from each other, which is achieved by virtually clipping nodes to different 2D planes. Because this is initially the only sensible and available information about the position of nodes, additional information are added. This is achieved by a novel method to generate spacial close groups of nodes. It makes sense following the explored idea of defining closeness of edges by not only position, but also various data attributes [12] and in this case also nodes.

Using the previously mentioned importance values (4.1) it is possible to group these nodes according to their similarity. For this purpose the Edge Bundling method KDEEB from [12] (described in 3.3.1) is adopted and modified. The nodes are iteratively advected according to their importance vector. Therefore, instead of calculating a single density map, several maps are calculated, one for each dimension of the importance vectors. Modifying 6 with 11 provides us the following equation for the density values:

$$\rho_k(x) = \sum_{i=1}^N \left( \iota_k \cdot \int_{y \in e_i} K \left( \frac{x-y}{h} \right) \right) \quad (12)$$



For  $K$  the Epanechnikov kernel  $K(x) = 1 - \|x\|^2$  is chosen as suggested by [12]. This leads to a density vector function  $P : \mathbb{R}^3 \rightarrow \mathbb{R}^n$

$$P(x) = (\rho_1(x), \dots, \rho_n(x)) \quad (13)$$

With  $P$  it is possible to calculate similarities between nodes and positions in the density map using the scalar product of both vectors.

The following equation is designed to act as an similarity based density function  $\psi : \mathbb{R}^2 \rightarrow \mathbb{R}$

$$\psi(x) = \|P(x)\| \cdot \left( \left\langle \frac{I}{\|I\|}, \frac{P(x)}{\|P(x)\|} \right\rangle - \tau \right) \quad (14)$$

Since the importance values  $\iota$  and thus the accumulation of the density vectors (13) are always positive, the resulting values for the scalar product of these vectors can only be positive too. If each vector is normalized before, the scalar product is between 0 and 1 and is taken as a similarity measure between these two vectors.

With the desired feature in mind that similar nodes attract each other while dissimilar nodes repel each other,  $\tau$  is added. This variable acts as a threshold for similarity attraction. If the similarity is smaller than  $\tau$  the value is negative and indicates a less desirable position at the density map. Finally the length of the density vector  $\|P(x)\|$  scales the attraction/repulsion, to assign a higher value to locations on the map with many similar nodes.

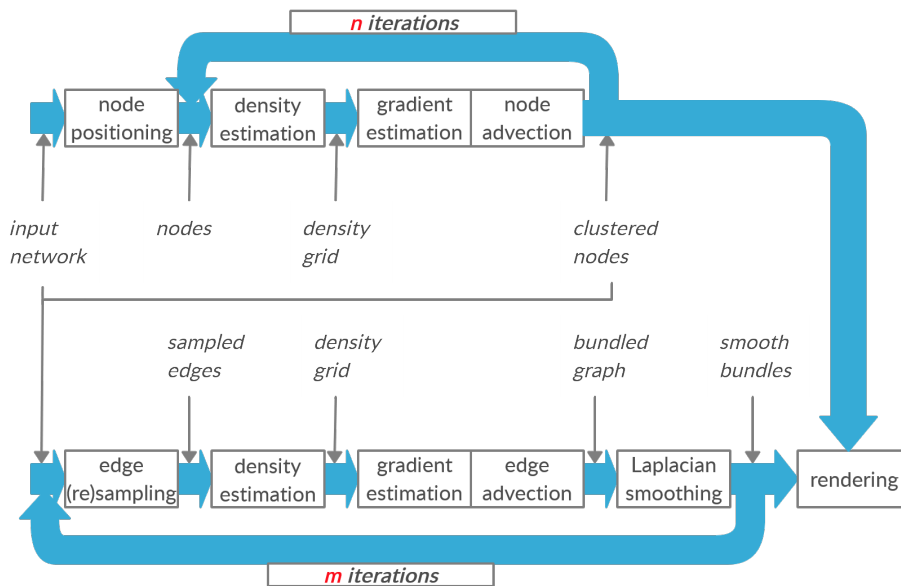
In analogy to (15) the operator  $BI : \mathbb{R}^3 \rightarrow \mathbb{R}^3$  is defined as the solution of the following ordinary differential equation:

$$\frac{dx}{dt} = \frac{h(t)\nabla\psi(t)}{\max(\|\nabla\psi(t)\|, \epsilon)} \quad (15)$$

With this operator the nodes of a neural network will be iteratively displaced according to their importance vector, creating groups of similar nodes in 3D space. To ensure that the nodes remain on the 2D plane defined for their layer, the nodes can simply be projected back onto these planes. If the planes are axis-aligned, it is possible to simply skip the specific displacement dimension.

Edges are approximated similar to KDEEB (3.3.1) by sampling points along their path. Instead of bundling these edges together by unambiguously advecting their samples, the importance vectors of the edges are utilized with the method previously described for advecting nodes. While the smoothing phase in KDEEB is skipped for nodes, it is kept for edges in each iteration after the sample advection phase.

The whole process is illustrated in 13. The neural network with its importance values is the required input. For rendering both the clustered nodes



**Figure 13:** Network edge bundling pipeline

and the smooth bundles from the edge processing is combined for the rendering phase.

Instead of two separate steps for gradient estimation and advection (like in 10) the process is combined. To use the relative density as shown in (14) to calculate the gradient, information about each node or sample must be accessed. Since the gradients are different for each element, it is not possible to calculate a universal gradient map.

Like the method on which it is based, most parts of the process can be parallelized to a high degree and present a feasible option especially with the latest hardware improvements in graphics cards.

### 4.3 Rendering

The network is represented as a graph with nodes and edges in 3D. Spherical 3D-Impostor are used for the nodes, while the edges are rendered using their samples to create elliptical 3D-Impostor. A special transparency method 3.4.2 is applied to the primitives to mitigate occlusion problems that are common in 3D visualizations.

### 4.3.1 Cuboid Impostor for Ellipsoids

While nodes can be rendered using the sphere impostor described in 2.3, this method is not originally practical for edges. Regarding the information gained for edges through the previous processing, the input is a number of samples for each edge. Rendering these samples as spheres is not providing the desired feature for an edge. Therefore the use of ellipsoids, which bridge the distance between the samples seems more appropriate than the use of concatenated spheres, .

A desired primitive for edge samples would be a tube-like object whose concatenation results in string-like objects representing edges. Tubes enable varying the thickness for different edges but need additional work in smoothly merging one end of a tube to the next. This problem leads to the consideration of a different type of primitive. Overlapping ellipsoids seem to provide smoother transitions between successive primitives.

Cuboids are created by stretching a cube impostor along the vector between two successive edge samples. By using a matrix transformation to transform information between world coordinates and the stretched space within the cuboid, the stretch vector is axially aligned. This makes it easy to modify the ray-sphere intersection calculation 3.4.1 by simply scaling the ray direction vectors along the stretched axis.

The equation (7) is modified by the axially aligned ellipsoid radii  $\vec{e} \in \mathbb{R}^3$ :

$$\underbrace{\left(\frac{\vec{d}}{\vec{e}} \cdot \frac{\vec{d}}{\vec{e}}\right)}_a t^2 + 2 \underbrace{\left(\frac{\vec{f}}{\vec{e}} \cdot \frac{\vec{d}}{\vec{e}}\right)}_b t + \underbrace{\frac{\vec{f}}{\vec{e}} \cdot \frac{\vec{f}}{\vec{e}} - r^2}_c = at^2 + bt + c = 0 \quad (16)$$

$\vec{d}$  and  $\vec{f}$  are component-wise divided by the ellipsoid radii vector  $\vec{e}$ . The solution of the equation is analogous to the original (8).

### 4.3.2 Importance dependent Transparency

To highlight more important parts of the neural network, the transparency method described in 3.4.2 is modified to include importance values (11). This should reduce the effect of self occlusion and make the important parts of the neural network more visible, while hiding unimportant parts.

For calculating the opacity  $\alpha$  of a fragment for an element (node or edge) the following equation is used:

$$\alpha = g((1 - o_{base}), g(o_{importance}, \iota) \cdot g(o_{depth}, d) \cdot o_{density}) \quad (17)$$

With  $g : \mathbb{R}^2 \rightarrow \mathbb{R}$ :

$$g(x, y) = 1 - x + x \cdot y \quad (18)$$

This function effectively scales the influence of  $y$  by  $x$ .

$\iota$  is the importance value and can either be the overall importance or any value of the importance vector  $I$  (11) to highlight either the overall importance or the importance for the relevant output node of the neural network as described in 4.1. The rate, at which the importance is factored in  $\alpha$ , is defined by  $o_{importance} \in \mathbb{R}^+$ . The second part is the depth relative to the camera of the current view  $d$  and its influence is defined by  $o_{depth} \in \mathbb{R}^+$ .  $o_{density} \in \mathbb{R}^+$  is defined as the density of an element at the fragment position. This value is generated by calculating the intersecting portion of a ray starting at the camera position and passing through an object. Is the fragment showing the center of a sphere or ellipsoid, the normalized value will be the greatest at 1, while the value at the edge of a sphere or ellipsoid tends towards 0.  $o_{base}$  simply defines a base opacity for every element independent of any factors.

Instead of blending the different color values based on  $\alpha$ , the minimal color value for each RGB-component defines the resulting screen pixel:

$$C_{screen} = \min(c_0 \cdot \alpha_0 + (1.0 - \alpha_0), \dots, c_n \cdot \alpha_n + (1.0 - \alpha_n)) \quad (19)$$

The color  $c$  is multiplied by each overlapping elements fragment  $\alpha$  at the current pixel position. The larger  $\alpha$  gets the stronger the original color  $c$  of an element is presented. With  $\alpha$  getting smaller, the resulting value trends toward 1 and is therefore blending in with the white background.

By choosing the minimum, the order of elements in the rendering pipeline does not have to be considered, which is otherwise required for regular alpha blending. According to the equation, fragments with a larger  $\alpha$  are generally prioritized.

## 5 Implementation

The whole visualization method is implemented in Python using OpenGL. For neural networks Keras API [5] is used with TensorFlow [1].

The input for creating a 3D representation is an existing trained neural network. This network is first processed in two steps and can be rendered in real-time afterwards. The first step is to calculate the importance values for the network parameter. For utilizing these values a modified version of edge bundling is applied to the edges and nodes and stored as a processed model. This processed model can then be rendered in the 3D simulation.

This chapter describes the implementation of the presented visualization method of this thesis.

## 5.1 Importance calculation

To determine the importance of nodes and edges for the performance of an existing neural network model, the method modifies its architecture, trains it on specific labeled sample data and reads importance values from the fine-tuned modified models. A set of specialized networks are created by employing feature extraction (see 2.1.1) from the layer of the original model and then fine-tuning (see 2.1.1) it on the added layers to learn the importance through backpropagation.

Each output node of the original neural network model (or every potential class predicted by the model in the classification task) adds one dimension to the importance vectors that have to be calculated. Therefore the training data set needs to be split depending on the output node in focus.

Using the MNIST data set (see 2.1.3) as an example, there are 10 possibilities of labels corresponding to the different numerical digits. A model predicting these labels from pixel data of images would usually have 10 output nodes with each representing one digit. MNIST training data would be split according to their label.

Training a model on only one possible outcome is not sensible, therefore we add a portion of every other class to the subset of training data and binary label the data as either 0 for the relevant class or 1 for another class, without differentiating between them. It is necessary to identify characteristics, that show that the image corresponds to the relevant class as well as those that indicate, that it does not correspond to the specific class. For every class the resulting split should be 50% samples of the relevant class and 50% a mix of equal amounts from the samples for every other class.

For every potential class the original model gets modified and the output layer is changed into two nodes predicting it either as the “relevant class” or as a “different class”.

The models are modified by adding a batch normalization layer (2.1.4) between every existing layer. Before training the new models on labeled sample data, every parameter of all original layers get fixated to ensure that the original values are evaluated by the added batch normalization layers without changing them.

In a fine-tuning (2.1.1) sense the new tasks are focusing on one output node each and the data are subsets of the original data. The features of the original model layer are preserved by freezing their parameter. Only the new layers are optimized. The process results in multiple specialized models and is basically a combination of fine-tuning and feature extraction.

Training this modified model on the corresponding train data split, results in an importance estimate of every node in the original model for the specific class. The importance values are taken from the added batch normalization layers. The  $\gamma$  parameter of these layers can be interpreted as the importance of this node and will be added to the importance vector as

mentioned in 2.1.4. After repeating this process for every class, the result for the MNIST example would be a ten dimensional importance vector (11) for every node, indicating the importance for correctly predicting the corresponding digit in the original model.

The resulting set of importance vectors for nodes is combined with the edge weights in the original neural network, to generate the edge importance values. For fully-connected dense layers, this is achieved by combining the absolute edge weight with the importance vectors of the related nodes.

## 5.2 Neural Network Processing

The extracted importance in addition to the corresponding model is the input for this part of the implementation. A modified version of Edge Bundling (see 3.3.1) is applied to this data. Nodes and edges are extracted from the model and combined with their corresponding importance values. First the nodes are clustered. The edges are then sampled between the new node positions and subsequently bundled together in regards to their similarity. The following chapters expand on the process in more detail.

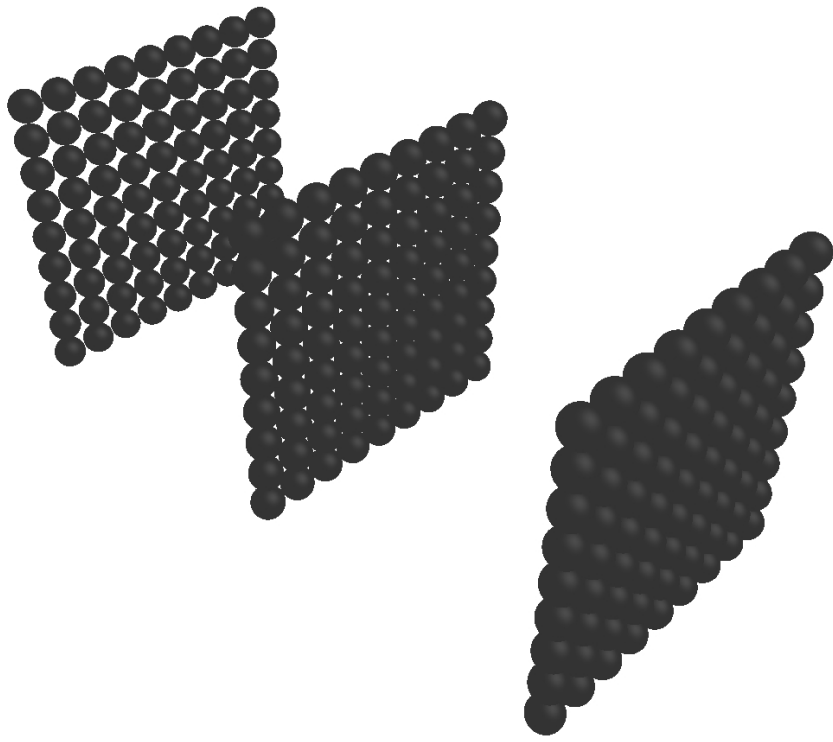
### 5.2.1 Grid

A key component of the Edge Bundling method is the calculation of density and gradients for the region in focus. To approximate the real density values, we divide the area into 3D cells of equal size. This is analogous to using textures in [12] to create density maps. This grid temporarily capture density vector information (13) from edge samples and calculate gradients between cells based on estimated density values.

Because of the memory-intensive nature of a uniform grid, a single grid ranging only between two layers is generated and simply reused for different layers by applying an offset to the access function of grid cells and resetting it in between usages. For simplicity a single grid ranging the entire network architecture is assumed in the following chapters.

### 5.2.2 Node Clustering

For the initial positions of nodes in a layer, the nodes are placed in a grid pattern (see figure 14) with equal distances in a defined square aligned to the  $x$  and  $y$ -axis. The resulting squares for each layer are set apart along the  $z$ -axis in equal distances. To ensure a related grouping of nodes along every layer regardless of the layer distance, the position of the nodes are



**Figure 14:** Nodes of a neural network as they are initially set up separated by layer.

projected to a single plane by disregarding their  $z$ -coordinate while accessing the grid. In the following steps the nodes are only displaced along the  $x$  and  $y$ -axis in order to keep them in the plane defined for their layer. The algorithm 1 shows the implementation of the node clustering method, which is described in chapter 4.2.

---

**Algorithm 1:** Node Clustering

---

**Data:** nodes, grid  
**Result:** clustered nodes

```

1 set initial node positions;
2 while bandwidth >  $\epsilon$  do
3   foreach node do
4     set radius to bandwidth;
5     apply node density to grid cells in radius;
6   end
7   foreach node do
8     calculate gradients from nearest grid cells;
9     move node in gradient direction by bandwidth;
10  end
11  reduce bandwidth;
12  clear grid cells;
13 end

```

---

The bandwidth  $h_{max}$  (10) for nodes starts at a fraction of the initially defined width of a layer plane and gets reduced by  $\lambda = 0.95$ , which tends to reduce the length of the displacement vector for node positions slowly enough to form groups of similar nodes.

An increasing similarity threshold  $\tau$  from (14) is chosen to prevent scattering groups of similar nodes too early. When choosing a larger threshold right from the beginning of this process, a larger proportion of nodes that are similar may end up in different location separated by dissimilar groups of nodes.

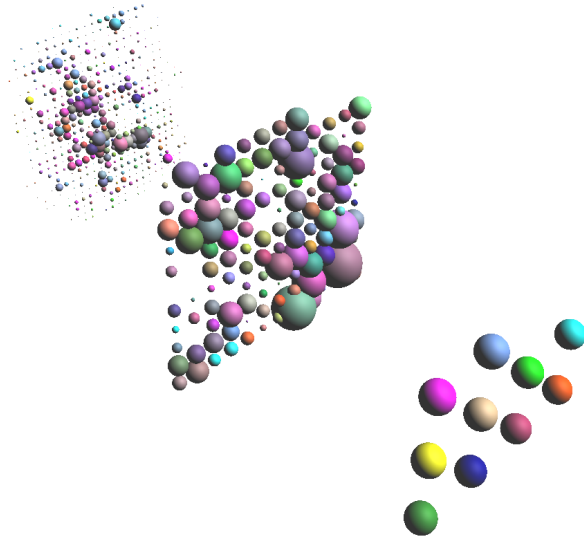
By increasing the threshold value antiproportionally to the bandwidth at each iteration, the nodes are first drawn to the center of the plane. Over time, as the dissimilarities begin to have repelling effects, the nodes begin to spread outward, while remaining closer to their similar neighbors, resulting in fewer but more distinct clusters of nodes.

An example for the clustered nodes of a neural network is presented in 15.

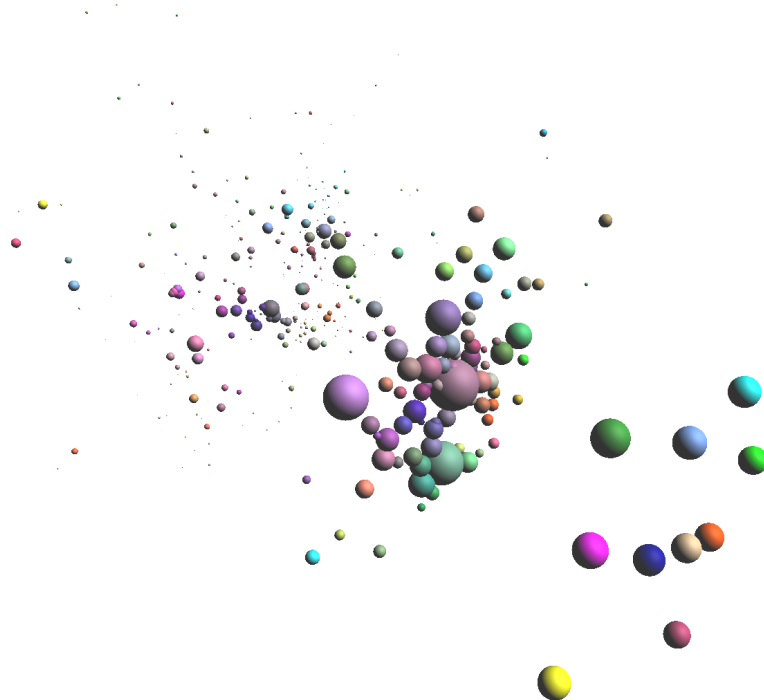
### 5.2.3 Edge Bundling

After the node clustering converges, the Edge Bundling process starts. To apply this method, the edges of the graph of the neural network must be





(a) Nodes of a neural network colored by their importance in the initial positions.



(b) Nodes after clustering by their importance similarity.

**Figure 15:** Comparison of initial node positions and post-clustering positions.

sampled. The number of samples for an edge depends on the length of the edge and changes in the advection process. The initial sampling and following resampling phases make sure that the distance between two neighboring samples of one edge stay constant. To ensure the grid (5.2.1) is sufficiently fine-grained for the desired sampling rate, the grid cell size is defined as a fraction of the distance between edge samples. The importance vectors of the edge samples are calculated by multiplying the edge weight with the importance vector linked to the edge.

The algorithm 2 shows the implementation of the edge bundling method, which is described in chapter 4.2.

---

**Algorithm 2:** Edge Bundling

---

**Data:** edges, grid  
**Result:** bundled edges

```

1 sample edges;
2 while bandwidth >  $\epsilon$  do
3   foreach edge do
4     foreach sample do
5       | set radius to bandwidth;
6       | apply sample density to grid cells in radius;
7     end
8   end
9   foreach edge do
10    foreach sample do
11      | calculate gradients from nearest grid cells;
12      | move sample in gradient direction by bandwidth;
13    end
14  end
15  repeat n times
16    foreach edge do
17      | smooth samples;
18    end
19  end
20  reduce bandwidth;
21  clear grid cells;
22 end

```

---

The similarity threshold  $\tau$  is changed in the same way as described with the node clustering and is increased antiproportionally to the bandwidth for edges. The bandwidth  $h_{max}$  (10) for edges starts at a fraction of the initially defined width of a layer plane and gets reduces by  $\lambda = 0.9$  similar to the value chosen in [12]. The bundling process stops when the bandwidth gets too small and the process converges and stops.

An example for the bundled edges of a neural network is presented in 16.

### 5.3 Neural Network Rendering

The processed network is rendered in 3D using the transparency technique described in 3.4.2, which was designed to utilize spheres. At its core the technique needs a volumetric primitive for it to be applied, but is not limited to spheres. The intersecting part of a ray passing through the primitive defines the transparency value. Therefore the technique is applied with two different primitives. On the one hand the nodes are represented as the classic spheres, on the other hand concatenated ellipsoids resemble edges. Using the more intricate ellipsoids over the spheres for edges provide a clear advantage. As demonstrated in figure 17, spheres need larger or more primitives to represent a continuous smooth line compared to the ellipsoids.

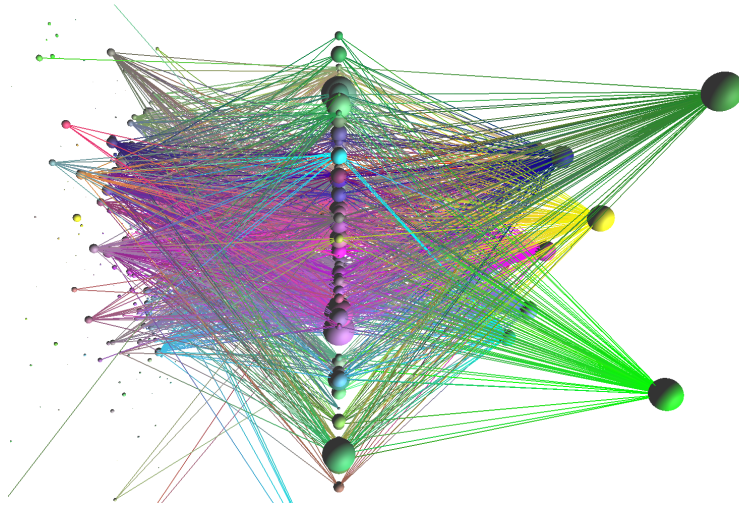
Ellipsoids are created for every two consecutive samples of an edge. To create smoother lines using these impostor, the ellipsoids are stretched along the vector between two samples. For straight lines this yields no problems and works fine with moderate curvature. The relations between primitives and sample positions are shown in 18.

The processed neural network can be visualized in different ways. For every class one distinct color is configured. It is possible to highlight the architecture related to a single class, all classes at once or by overall importance. An example of highlighting a class in the final visualization is shown in 19.

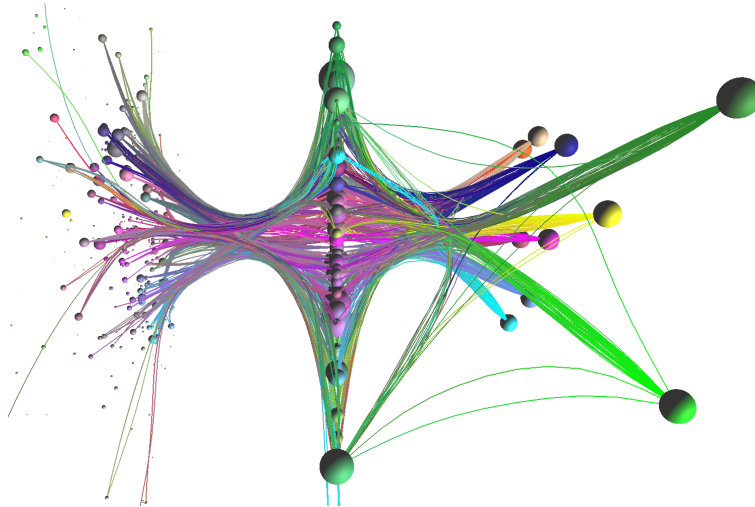
## 6 Analysis and Evaluation

The lack of established 3D visualization techniques highlighting the trained parameters makes it difficult to compare the method introduced in this thesis. Therefore evaluation is split in two parts. In the first part the underlying processing procedure of neural networks for extracting information is verified. The second part centers around validating the visualization in regard of its potential in identifying common attributes associated with learned parameter.

For the evaluation multiple neural networks were trained and used. The evaluation focuses on fully-connected layers. This type of layer is represented in many neural networks, but is describes as the “bottleneck in terms of memory consumption” [4] in many cases. Following model architecture will be analyzed in the next chapters:

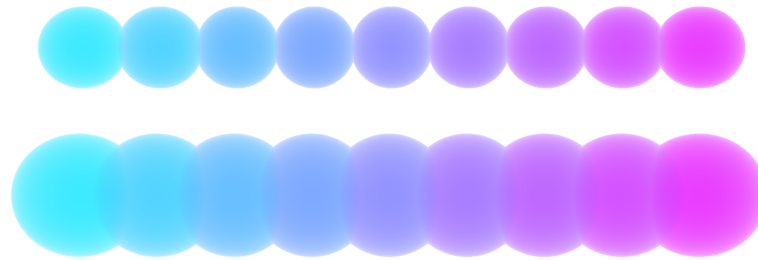


(a) Edges of a neural network colored by their importance in the initial positions.



(b) Edges after bundling by their importance similarity.

**Figure 16:** Comparison of initial edge sample positions and post-bundling positions.

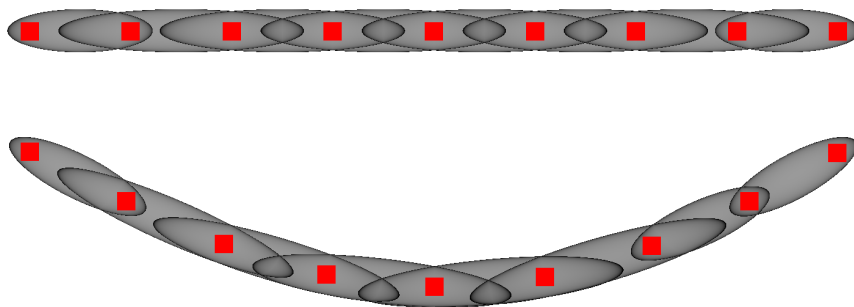


(a) Edge samples rendered as spheres in different sizes.



(b) Edge samples rendered as ellipsoids in different sizes.

**Figure 17:** Size comparison of sphere impostor and ellipsoid impostor using the same number of samples for an edge.



**Figure 18:** Ellipsoid impostor with highlighted sample positions in a straight line and with curvature.



(a) Front side of the processed model. (b) Back side of the processed model.

**Figure 19:** Processed model rendered using the transparency method and highlighting a selected class from different points of view.

**Model A** is trained on the MNIST database (see 2.1.3) and predicts the 10 different classes. The input layer consists of the 784 input nodes followed by one hidden layer with 128 fully-connected nodes. The output layer is also fully-connected and represent each class with one node (784-128-10). The graph representing this network consists of 101,632 edges and 922 nodes. The network is trained with the default settings for Dense layer in Keras. The hidden layer applies the RELU activation function, while the output layer uses Softmax. The batch size is set to 128 and the training epochs are set to 15, resulting in an accuracy of 97.82% on the test data.

## 6.1 Importance

To validate the calculated importance of a model (see 5.1), pruning (see 3.2.1) is applied. The usual verification process consists of pruning the models and calculating the accuracy then pairing them with the rate of pruned parameters like the parameter efficiency [24]. By pruning weights on a trained model by itself is not performing well on improving accuracy [23], therefore most pruning must be fine-tuned after pruning to regain or even improve the accuracy of the original model. Decreasing accuracy on pruned models does not need to be addressed for this thesis. Pruning is solely used to verify the importance measure, which is used to visualize the unchanged original model.

This fact paired with the missing experimental standardization makes the comparisons to existing methods difficult [2]. Because of the resulting fragmentation in reported data, the evaluation focuses on a simple form of pruning by only using the edge weights. A more extensive evaluation could be done in further research, but is not a high priority, as there seems to be no established measurements to which approach performs the best and strongly depends on applications and requirements [4].

In order to determine whether the importance is an accurate measurement,

the parts of the neural network are pruned depending on these values and the change in the accuracy of the model is observed. In the following presented results every pruned model is not fine-tuned after pruning, which is otherwise done for many of the presented results in related works. All accuracy measurements from the figures of this section are based on test data, which was not directly used for training these networks.

### 6.1.1 Overall Importance

For the following analysis an overall importance value for each node is calculated and analyzed. The measure of importance over all classes is defined as an average of all values of the Importance vector:

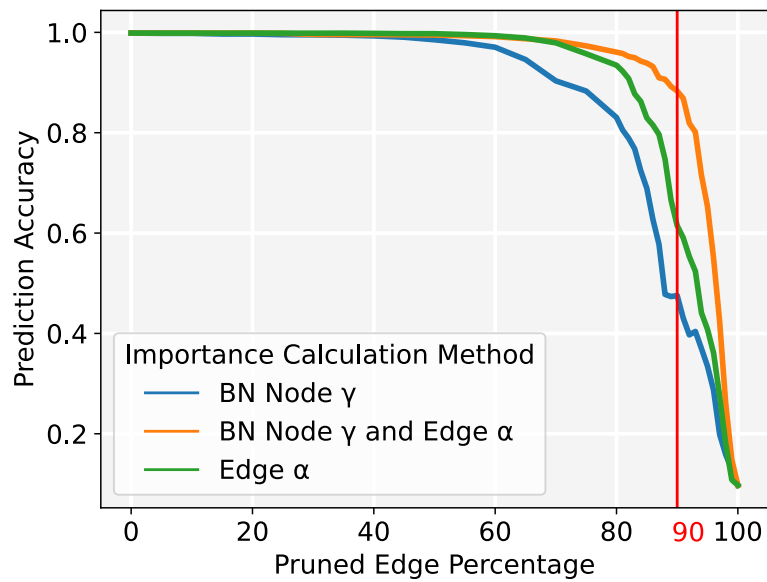
$$l_{overall} = \frac{1}{n} \sum_{i=1}^N l_i \quad (20)$$

For the evaluation three types of importance calculations are compared. The first is simply using the weight of the edges as it is already common practice in some pruning methods. For the second one the scaling value is extracted from the added batch normalization layer as described in 5.1. And lastly a combination of both as a product of the two values.

Using the weight  $\alpha$  of an edge should result in a decent compression rate, as it is already proven to be a good indicator for the importance on its own. The use of the  $\gamma$  value, which is not part of the original evaluated model is untested to the best of my knowledge. In 3.2.2 the batch normalization layers are part of the original model and are proven to be good indicator for the importance. Therefore the assumption that these values should also result in good predictions of importance is plausible. One disadvantage of using  $\gamma$  alone is that you can only prune complete nodes with all edges together, instead of preserving sporadically distributed important edges. A combination of both enables application of the importance on single edges as well. The hypothesis is that this combination should provide better results as the use of  $\gamma$  alone.

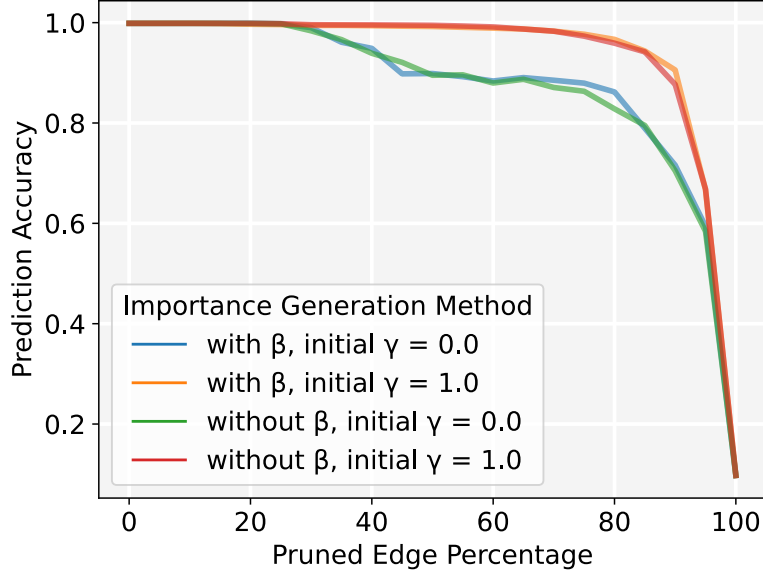
These three methods are applied on **Model A** and then compared. The result is presented in 20. The combination outperforms the separated methods. At a pruning rate of 90% the pruned model for the combination achieves an accuracy of 87.52%, while using the edge weights drops the accuracy to 61.66% and the use of the  $\gamma$  value alone drops it further to 45.81%.

The lower performances resulting from only using the  $\gamma$  value of the batch normalization layer could be explained by the pruning of complete nodes instead of single edges, but still provides reasonably good results. The performance changes as expected with using the edge weights. It is interesting that the improved preservation of performance from the combination validates this method as superior at least for the tested type of neural network.



**Figure 20:** Comparison of three importance calculation methods by the accuracy of pruned models. The parameters are pruned in order of lowest importance. Blue using only the  $\gamma$  value from the related batch normalization layer, Green using only the weight of an edge, Orange using a combination of both values.





**Figure 21:** Comparison of importance generated with different settings for batch normalization layer parameter. Blue and Orange use centering with  $\beta$  while for Green and Red this parameter is disabled.

To check the effect of the different settings for the added batch normalization layer, importance data is generated for **Model A** in four possible combinations of two distinct settings. The equation (4) consists of two major parameters  $\gamma$  and  $\beta$ . The effect of  $\beta$  values is tested, which is adding a trained value to an output regardless of the activation of the inputs. It is assumed that using this value should not have a positive effect on the generated importance values, since they are not directly affected by the variation of the input values and can be considered as being static.

The  $\gamma$  values, which are later extracted directly from the trained model, are a key factor of the whole concept. While it is not practical to disable these values, it is possible to initialize them differently. Because of the way learning of neural networks works, the hypothesis is that zero values could get stuck in training and important nodes and edges are potentially ignored. This should lead to worse results than the alternative initialization with a scale of one.

Figure 21 verifies the expected results by the same method used previously.  $\beta$  has no significant effect on the tested **Model A** while  $\gamma$  seems to provide better results on initialization with 1.0.

The importance values are extracted from parameters of modified and trained

network models. These values can therefore be seen as predictions based on the training data and are not perfect measurements. As with most parameters of a neural network the training process can be modified with regularizer (see 2.1.2). As suggested in [23] enforcing of sparsity during training through regularizers barely affects the performance and leads to increased generalization accuracy. They also stated the use of specifically  $L_1$  sparsity leading to smoother pruning and little accuracy loss.

To enforce a more sparse sets of  $\gamma$  values for a network, regularizer are added to their training and tested.

Figure 22 compares the effect of training with the different regularizers. In **(a)** any of the tested regularizer seem to bring a similar improvement (at 90% pruning approx. 88% accuracy), but without using one it seems to perform worse (at 90% pruning approx. 65% accuracy). In **(b)** the effect becomes more clear. Using no regularizer has the opposite effect of identifying important values and leads to pruning important parts of a network early on.

One explanation for this can be, that the modified network becomes a completely different network with many heavy scaling factors that are changing the inputs of following layers completely, despite using fixed parameters for the different layers. With regularizers the scaling factors are more limited, which would probably lead to an increased reliance on the fixed parameters for the optimization.

For this thesis and the presented 3D visualizations, the importance is calculated using  $L_1$  regularization for more reliable importance values.  $L_1$  is also used for the previously shown plots in this section.

### 6.1.2 Class Importance

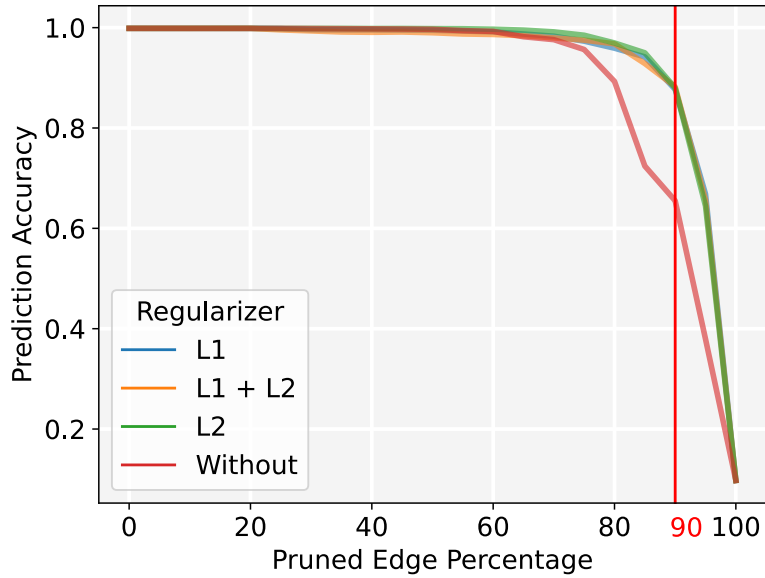
The importance values are calculated for each class. To verify how accurate this measurement is, applying pruning and simply observing the accuracy of the original model is not sensible. Instead the accuracy of predicting a single class is used. The predictions are categorized as either *important class* or *different class*. With this categorization the original test data is highly unbalanced with its many classes. Therefore the balanced accuracy is calculated:

$$ba = \frac{tp\ rate + fp\ rate}{2} \quad (21)$$

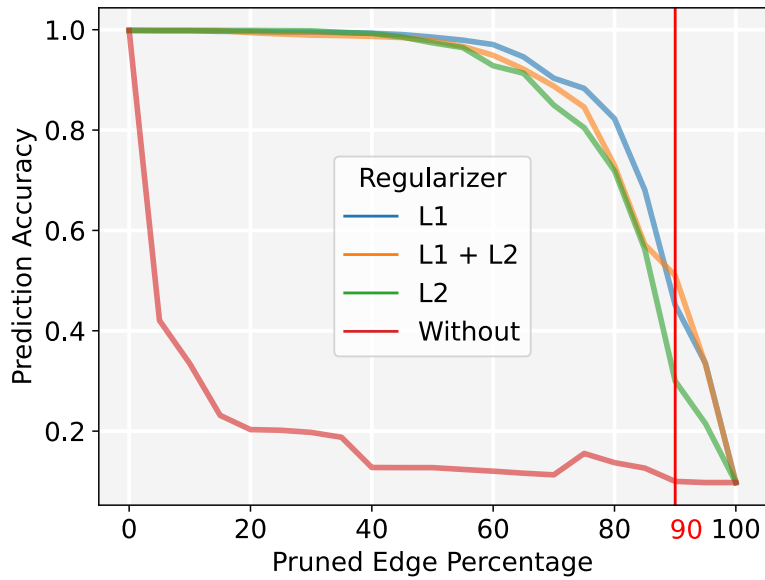
$$tp\ rate = \frac{Positives\ correctly\ classified}{Total\ positives} \quad (22)$$

$$fp\ rate = \frac{Negatives\ correctly\ classified}{Total\ negatives} \quad (23)$$

These equations are taken from and further explained in [8]. The balanced accuracy is used to measure how well a model predicts input data as the

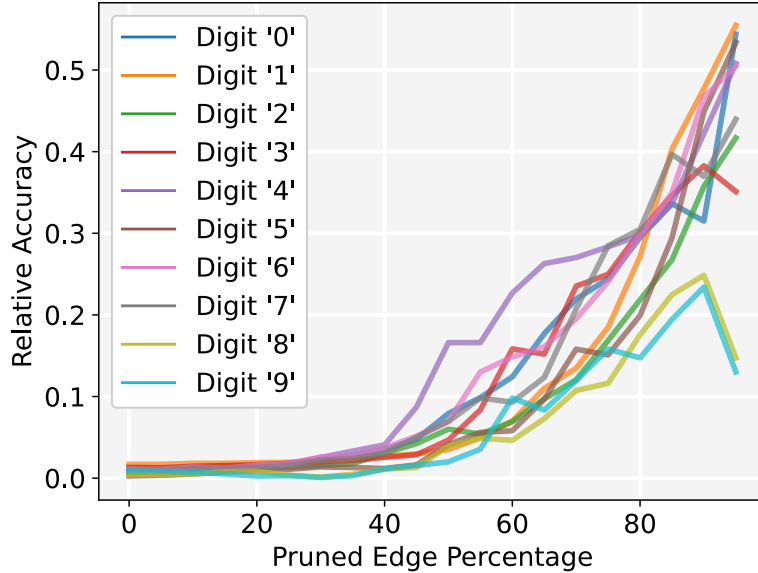


(a) Tested models using the  $\gamma$  value from the related batch normalization layer combined with the edge weight as importance for pruning.



(b) Tested models using only the  $\gamma$  value from the related batch normalization layer as importance for pruning.

**Figure 22:** Comparison of four regularizer used in the added batch normalization layer by the accuracy of pruned models. The parameters are pruned in order of lowest importance. Blue uses  $L_1$ , Green uses  $L_2$ , Yellow uses  $L_1$  combined with  $L_2$  and Red uses none



**Figure 23:** Performance comparison of neural network models pruned depending on the importance to the relative class. The relative accuracy is the difference between the balanced accuracy for predicting the class and overall accuracy of a network.

*relevant class* or as a *different class* with equal influence of both types.

To verify if the measure of importance for specific classes is accurate, we use the same method from 6.1.1. For every class the neural network parameters are pruned in order of the importance values related to this class. From these pruned models the balanced accuracy regarding the corresponding class is compared to the overall accuracy in predicting every distinct class. When pruning parameters that are important for a related class, the accuracy for a class should be preserved to a certain degree while the overall accuracy would degrade more intensively. Therefore the difference between those two accuracy measures should increase.

In figure 23 these differences are shown for every class. The *Relative Accuracy* value being positive means that the balanced accuracy for predicting the class correctly is greater than the overall accuracy. The difference is steadily increasing, until the value comes close to 100%. This trend proves, that these importance values at least resemble the real importance to some degree, because the parameters more relevant to the class are more likely to be preserved.

## 6.2 Bundling Comparison

The visualization contains information about the parameters and their importance for the predictions of a neural network. The logical conclusion is that vastly different neural networks should also differ in their representation. If the processed models gain distinct identifiable characteristics, it becomes possible to abstract information by looking at their visualization and recognizing these characteristics.

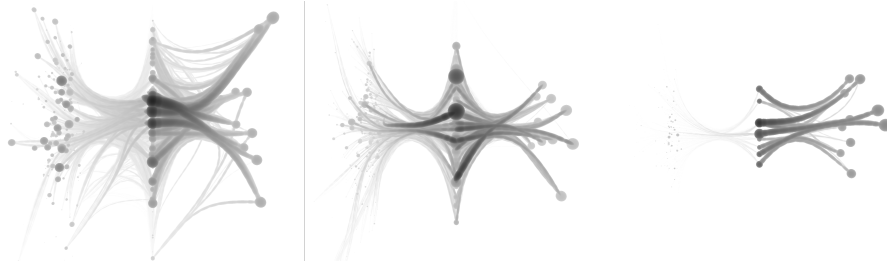
With this notion in mind multiple models are created for the current part of the evaluation. All models are identical in their initial architecture, which is described as **Model A** (see 6). The only difference between these models is the training process modifying the parameter values that are ultimately presented in the visualization method. When using different training techniques, different characteristics should emerge. By using the exact same architecture for the models we reduce interferences. Three different models are created to analyze the differences:

- The **untrained model** received their randomly initialized parameter and is not further changed.
- The **basic model** is trained using the regular training data. With no special settings for their fully connected layers.
- The **regularized model** is also trained using regular training data. The difference lies with the introduction of  $L_1$  regularization on the fully connected layers during training.

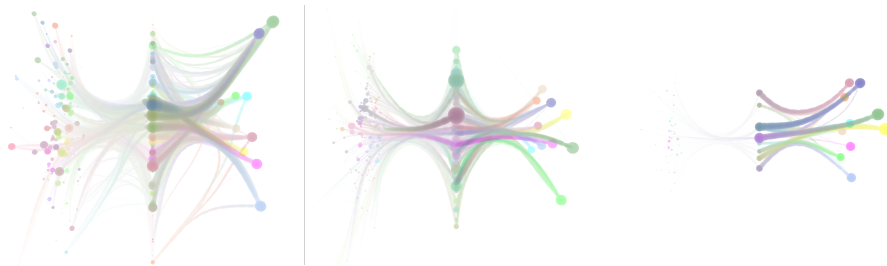
The expectation for the untrained model should be a lack of structure resulting from the random parameter values, while the basic model should learn to generalize to achieve its high performance. Lastly the regularized model should represent a certain sparsity resulting from its  $L_1$  regularization.

Figure 24 shows that a certain degree of distinctions can be made. For the untrained model the nodes are more spread out, which potentially indicates generally greater differences between the nodes and a lack of generalization. On the other hand the trained models seem to result in nodes being closer to each other. The effect can be explained by assuming a node, which is balanced in its importance for all classes. This node would naturally be more likely to be attracted by every other vector, because their scalar product results in greater values (see (14)). These generalizing nodes would be drawn towards the center while also advecting surrounding nodes to the center as a consequence. One characteristic of increased density can be identified and assigned to generalization.

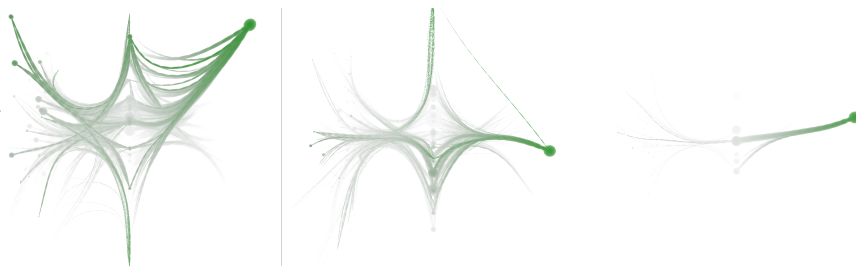
The sparsity is another attribute, which seems to be identifiable. Figure 25 shows more extreme bundling resulting in more isolated groups of



**Figure 24:** Comparison of three different processed neural networks. Their parameter are highlighted according to their overall importance. On the left is an untrained neural network, in the middle is the network trained with basic settings and on the right trained using  $L_1$  regularization for the original layer.



**Figure 25:** Comparison of three different processed neural networks. Their parameter are highlighted and colored according to their importance vector. On the left is an untrained neural network, in the middle is the network trained with basic settings and on the right trained using  $L_1$  regularization for the original layer.



**Figure 26:** Comparison of three different processed neural networks. Their parameter are highlighted and colored according to their importance regarding a single class. On the left is an untrained neural network, in the middle is the network trained with basic settings and on the right trained using  $L_1$  regularization for the original layer.

nodes and edges and becomes clear in figure 26. An explanation could be the sparsity in the resulting importance vector leading to more segregation in the advecting process. The intra-group similarity increases, while the cross-group similarity becomes smaller leading to the observed result.

## 7 Conclusion

The visualization method of neural networks with its underlying processing seems to produce promising results. While the evaluation lacks direct comparisons it still manages to prove its potential on basic problems regarding neural networks. The novel approach presented in this thesis combines the research field of network optimization with Edge Bundling and specific rendering techniques in an intricate way.

More extensive research needs to be done to draw solid conclusion about the presented method as it is only tested on a fairly simple subset of neural networks. How the insights gained from this work could be used is explained in the next chapters.

### 7.1 Usability

The introduced method of generating importance values for an existing neural network is promising regarding Network Optimization. It could be used to identify problematic output nodes of trained neural networks and as a pruning concept itself, which needs further evaluation to be compared to existing methods. Another use-case could be in feature extraction for single classifications by simply pruning parameters unnecessary for specified classifications, which is proven to work in the evaluation part of this thesis. Instead of using complete layers singular parameters could be directly chosen.

There is potential in a visual tool that can be used for analysing neural network parameters. The visualization could be used in an interactive tool for targeted pruning of parameters, although the computation time of the bundling process is not to be neglected. While the visualization of the resulting 3D-model itself is real-time capable, the creation is not for moderately to large neural networks. But an analysis tool could still be used with prepared 3D-models, on which pruning parameters can be done in real-time. Directly observing the performance for sample data is also possible, depending on the inference time of the original neural network model. The usability of such a tool in a professional setting ultimately needs to be evaluated by a group of experts. As for an educational sense it is arguably sensible to find potential use. Comparing generated 3D models might be a

feasible way of getting insights about potential issues of a model like low generalization, although this needs a more extensive analysis on a wide range of models.

## 7.2 Future work

The conceptualized method for extracting importance values uses subsets of data to create importance values for these specific data sets. Instead of splitting data regarding different classifications, it is possible to split the data for different reasons. In cases where you have large amounts of artificially created data and a small set of real world data, it would be possible to train the network on the complete data and select a subset of the created model in regards to the importance for the expected data during inference. Another aspect similar to the problem of poor generalization is redundancy in neural networks. By bundling similar nodes this issue comes to mind and there may lie untapped potential in identifying and reducing redundancy instead of simply pruning by looking at the similarity of parameters in the importance values.

For applying the method to extremely large neural networks optimizations like using octrees instead of a uniform grid for the bundling process would be beneficial. Another way to improve performance would be to increase the rendering time of the visualization by combining closely bundled samples or edges instead of rendering the many overlapping edge sample primitives.

So far only fully-connected layer are represented, but because of the nature of the importance calculation the representation of various types of layers seems feasible. In future works this method can be expanded for other prominent layer types like convolutional layers.

The introduced visualization technique with the importance estimation for neural network parameters proved to be a promising method, but needs a lot more verification and refinement. In this thesis it is shown, that there is potential in this type of visualization for neural network parameters, which is worth further investigation.



## References

- [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. Wattemberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
- [2] D. W. Blalock, J. J. G. Ortiz, J. Frankle, and J. Gutttag. What is the state of neural network pruning? *ArXiv*, abs/2003.03033, 2020.
- [3] Y.-C. Chen. A tutorial on kernel density estimation and recent advances, 2017.
- [4] Y. Cheng, D. Wang, P. Zhou, and T. Zhang. A survey of model compression and acceleration for deep neural networks. *CoRR*, abs/1710.09282, 2017.
- [5] F. Chollet et al. Keras. <https://keras.io>, 2015.
- [6] L. Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *Signal Processing Magazine, IEEE*, 29:141–142, 11 2012.
- [7] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.
- [8] T. Fawcett. Introduction to roc analysis. *Pattern Recognition Letters*, 27:861–874, 06 2006.
- [9] E. Haines, J. Günther, and T. Akenine-Möller. *Precision Improvements for Ray/Sphere Intersection*, pages 87–94. Apress, Berkeley, CA, 2019.
- [10] A. W. Harley. An interactive node-link visualization of convolutional neural networks. In *ISVC*, pages 867–877, 2015.
- [11] P. Hermosilla, P. Vázquez, A. Vinacua, and T. Ropinski. A general illumination model for molecular visualization. *Computer Graphics Forum*, 37(3):367–378, 2018.
- [12] C. Hurter, O. Ersoy, and A. Telea. Graph bundling by Kernel Density Estimation. In *EUROVIS 2012, Eurographics Conference on Visualization*, volume 31, pages pp 865–874, Vienna, Austria, June 2012. Wiley.

- [13] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.
- [14] C. Käding, E. Rodner, A. Freytag, and J. Denzler. Fine-tuning deep neural networks in continuous learning scenarios. In *ACCV Workshops*, 2016.
- [15] M. D. Kissner. Hacking neural networks: A short introduction. *ArXiv*, abs/1911.07658v2, 2019.
- [16] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.
- [17] A. Lambert, R. Bourqui, and D. Auber. 3d edge bundling for geographical data visualization. In *2010 14th International Conference Information Visualisation*, pages 329–335, 2010.
- [18] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. 1998.
- [19] Y. LeCun, C. Cortes, and C. Burges. Mnist handwritten digit database. *ATT Labs [Online]*. Available: <http://yann.lecun.com/exdb/mnist>, 2, 2010.
- [20] A. LeNail. Nn-svg: Publication-ready neural network architecture schematics. *Journal of Open Source Software*, 4(33):747, 2019.
- [21] G. Li, C. Qian, C. Jiang, X. Lu, and K. Tang. Optimization based layer-wise magnitude-based pruning for dnn compression. In *IJCAI*, 2018.
- [22] Z. Li and D. Hoiem. Learning without forgetting. *CoRR*, abs/1606.09282, 2016.
- [23] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang. Learning efficient convolutional networks through network slimming. *CoRR*, abs/1708.06519, 2017.
- [24] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell. Rethinking the value of network pruning. *CoRR*, abs/1810.05270, 2018.
- [25] A. Nagpal. L1 and l2 regularization methods, 2017.
- [26] J. Rogawski. Visualisierung von molekülen durch bewegte bilder, 2016.
- [27] V. Sanh, L. Debut, J. Chaumond, and T. Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *ArXiv*, abs/1910.01108, 2019.

- [28] M. Tarini, P. Cignoni, and C. Montani. Ambient occlusion and edge cueing for enhancing real time molecular visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1237–1244, 2006.
- [29] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. Learning structured sparsity in deep neural networks. *CoRR*, abs/1608.03665, 2016.
- [30] W. Zhao. *Ranking and Sparsifying Edges of a Graph*. PhD thesis, USA, 2012.