FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Deep Learning Architectures for Vessel Segmentation in 2D and 3D Biomedical Images

am VRVis Zentrum für Virtual Reality und Visualisierung Forschungs-GmbH

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medizinische Informatik

eingereicht von

Mr Mario Zusag
Matrikelnummer 1427881

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Werner Purgathofer
Mitwirkung: Dr. Jiří Hladůvka
Dipl.-Math. Dr. Katja Bühler
MSc. Maria Wimmer

Wien, 16. August 2017

Mario Zusag

Werner Purgathofer

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Deep Learning Architectures for Vessel Segmentation in 2D and 3D Biomedical Images

at the VRVis Zentrum für Virtual Reality und Visualisierung Forschungs-GmbH

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Medical Informatics

by

Mr Mario Zusag
Registration Number 1427881

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Werner Purgathofer
Assistance: Dr. Jiří Hladůvka
Dipl.-Math. Dr. Katja Bühler
MSc. Maria Wimmer

Vienna, 16th August, 2017

Mario Zusag

Werner Purgathofer

Erklärung zur Verfassung der Arbeit

Mr Mario Zusag
Fasangasse 9, 2384 Breitenfurt

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 16. August 2017

Mario Zusag

Acknowledgements

I wish to express my sincere thanks to Prof. Werner Purgathofer, President and Scientific Director of the VRVis Research Center, as well as Dr. Katja Bühler, Head of the Biomedical Image Informatics Group, for letting me write my thesis at the VRVis and providing me with all the necessary facilities for the research.

I am also especially grateful to Dr. Jiří Hladůvka, under whose supervision and help I was implementing the deep learning architectures for vessel segmentation. He has taught me a lot in the field of machine learning, showed me many useful techniques for a practical deep learning approach and together with MSc. Maria Wimmer and MSc. David Major provided me with advice and valuable insights, whenever I needed them.

Abstract

The aim of this thesis is to describe deep learning approaches for vessel segmentation in 2 and 3-dimensional biomedical images and the results achieved from these approaches on specific sets of data. The first chapter introduces the objective of this thesis, describes the data, which was used for the training, gives a short overview of machine learning and covers some theoretical aspects of artificial neural networks and especially of convolutional neural networks. The second chapter describes methods that were used for achieving the segmentation in 2 and 3 dimensions, like preprocessing of the images, algorithmic approaches, and general project set-up. The third and final chapter focuses on the results of methods described in chapter 2, contains personal advice for future approaches for improving the algorithm's results and discusses the results. The thesis provides the theory, code snippets for the most fundamental part of the algorithms' implementations and shows graphical, as well as numerical results of the approaches.

Contents

CHAPTER 1

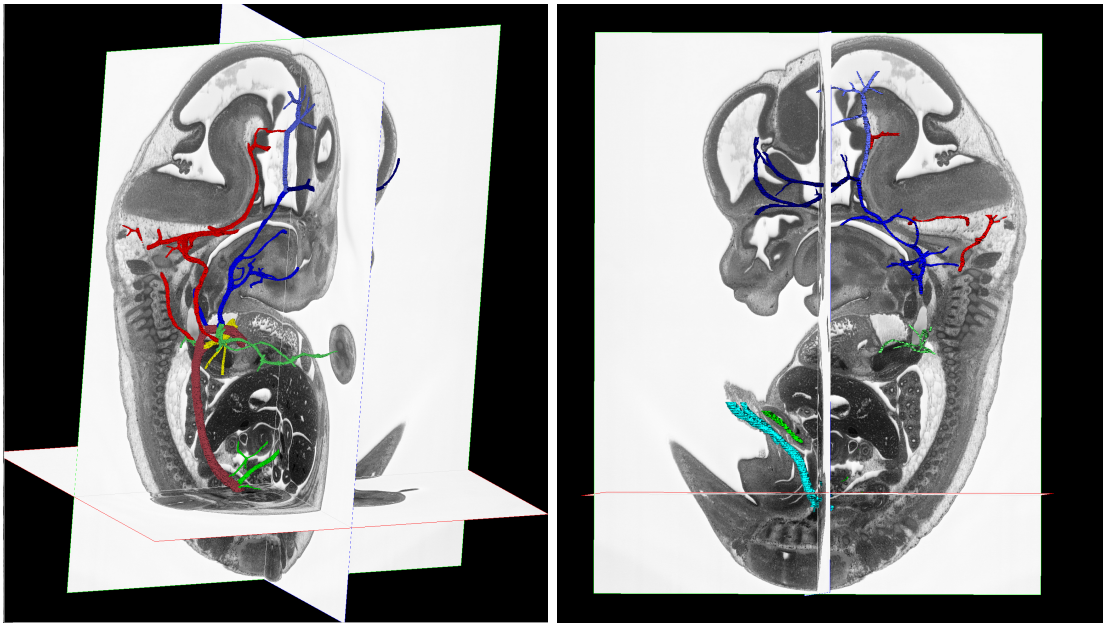
Introduction

This thesis' practical objective was to implement and assess different algorithmic approaches for detecting blood vessels in given biomedical images. The blood vessels were segmented by hand, which is not only a lengthy task, but also prone to errors. Automatic segmentation should tackle these two problems. Due to the high variability of images, a machine learning approach was used instead of a non-learning object or pattern recognition. The learning parts of the algorithmic approaches use *artificial neural networks* with *deep learning* aspects, the so-called *Convolutional Neural Networks*, which are described in section 1.4.

Section 1.1 describes the data used for training and testing, section 1.2 gives a short overview of machine learning and section 1.3 describes the theory of artificial neural networks.

1.1 Data

The images resulted from different *High-Resolution Episcopic Microscopy* (HREM) studies, a method for creating digital volume data of organic material by taking pictures of the sliced organic material [1]. The images for this thesis have a high spatial resolution with 2857 slices consisting of 2279×1753 pixel .jpg images and 3580 slices consisting of 2463×1691 pixel .jpg images. The labelling of the accompanying vessels was done by medical staff. They distinguished not only non-vessel tissue from vessel tissue but have also segmented different vessels in different colours. The distinction between different vessels was not part of this thesis' objective, only the segmentation of them. An example for illustration of the mice embryo body and the segmented blood vessels is depicted in figure 1.1.



(a) Mice embryo 3D (b) Mice embry 3D side view

Figure 1.1: Mice embryo sample.

1.2 Machine Learning

Deep learning, which is used in this thesis, is a specific kind of machine learning. The understanding of basic machine learning principals is crucial for understanding the algorithms and approaches used in chapter 2 because they differ from former and probably more intuitive methods like object and pattern recognition.

In 1959 Arthur Samuel defines machine learning as the "field of study that gives computers the ability to learn without being explicitly programmed" [2]. The learning process is extensively discussed by experts and has many definitions. The algorithms that are used to learn from given data are called *learning algorithms*. An operational and measurable definition for learning algorithms is given by T. M. Mitchell [3]:

> "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E."

The tasks can range from classifying objects, segmenting structures up to predicting numerical values and many other types. This section will focus on classification and segmentation. The data for these tasks must be processed in a quantifiable format, which is usually a vector $\vec{v} \in \mathbb{R}^n$ with n being the number of features we want to process [4].

One of the most famous classification tasks is the prediction of classes of the MNIST data set [5], which will also be used in the following sections to illustrate the learning procedure of neural networks. The MNIST data set consists of 70000 28×28 pixel images of handwritten digits, as shown in figure 1.2. The images are grayscale images, where each pixel has a value in the range of 0 (white) and 1 (black), as shown in figure 1.4.

Figure 1.2: 100 samples of the MNIST images

The functioning of a classification algorithm is the following: the algorithm is fed with an n-dimensional vector of real values and outputs a vector with the so-called *class scores*, i.e. the scores for each class the algorithm should be able to predict. This can be formulated as a simple function of the form:

$$f : \mathbb{R}^n \to \{y_1, \ldots, y_k\} \mid y_i \in \{0, \ldots, 1\}, i \in \{1, \ldots, k\}, k = \text{number of categories.} \quad (1.1)$$

The objective of the classification on the MNIST data set is to predict the digit that is displayed in each input image. As there are 10 digits, the mathematical formulation of the learning algorithm's objective function is $f : \mathbb{R}^{784} \to \{y_1, \ldots, y_{10}\} \mid y_i = 1$, if i is the correct label and $y_i = 0$ else. The images themselves are flattened and form a vector of size image $784 = 28 \times 28$ (i.e. height $\times$ width).

The performance P that should improve over the experience E can be measured by calculating the error rate or the accuracy of the learning algorithm. The selection of the best performance measure, i.e. error or accuracy function, is not always trivial but has large effects on the learning process. In section 1.3.4 some of the most common error or accuracy functions in machine learning are described.

In the case of the MNIST classification task, if the algorithm would just predict random digits, the error rate would theoretically be 90% and the accuracy would be 10% respectively, as there are 10 classes. There exist many different machine learning approaches

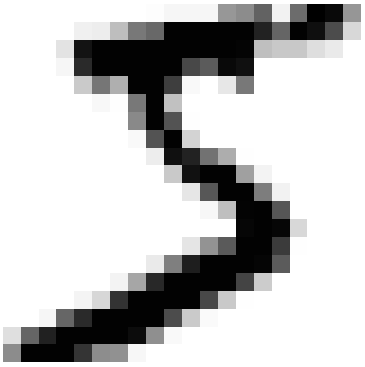


Figure 1.3: Handwritten 5

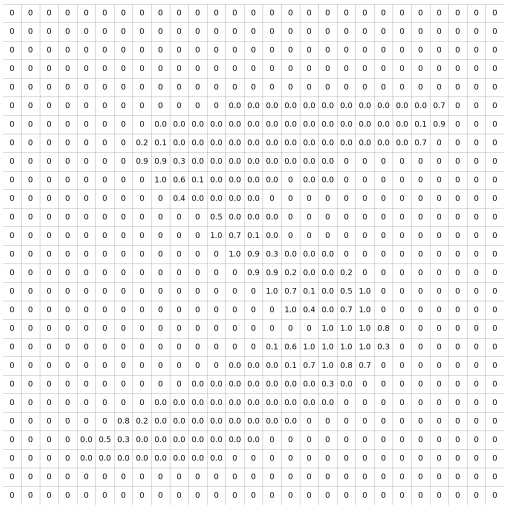


Figure 1.4: Pixel values of figure 1.3.

that have tried to achieve 100% classification accuracy, of which some are listed in [5]. The best solutions at the time of this paper yielded from convolutional neural networks, which are described in section 1.4, with an error rate of up to 0.2% as published in [6] for instance.

Another question that arises, is on which data the machine learning algorithm's performance should be measured. The usual approach is to split the original data into a *train set*, which is used for the learning part and a *test set*, which consists of data that the algorithm has not processed yet. If the algorithm predicts these unseen inputs well, we speak of a *generalized* learning algorithm. Without the ability to predict unseen data, the algorithm is just optimizing predictions on a given set of data and not learning to interpret the structures in general data [4].

If the learning algorithm performs well on the training set and is much worse on the test set, we speak of *overfitting*, which means that the algorithm learns too many features from the training set and cannot interpret general data. If the algorithm cannot learn useful features from the data, we speak of underfitting. An example for illustration is shown in the machine learning library for Python *scikit-learn* [7], which is used in figure 1.5. It shows an approximation approach of the nonlinear cosine function. The algorithm in the example tries to fit polynomials of different degrees through data points that approximate the cosine function.

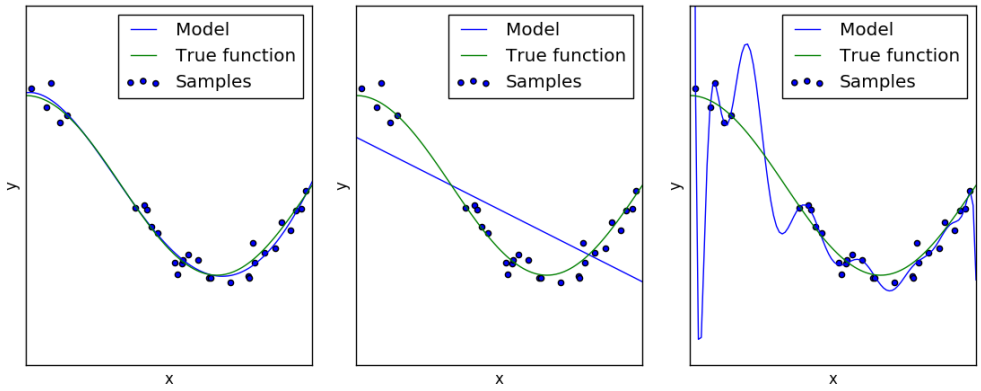
A polynomial is a mathematical function that takes the summation of the product of exponents and their accompanying coefficients. The degree of a polynomial is defined by its biggest exponent n. The general polynomial function of degree n is defined as:

$$P(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n, \quad n \geq 0 \text{ [8]} \tag{1.2}$$

In the case of the underfitting depicted in figure 1.5b the data points are fitted with a linear function, i.e. a polynomial of first degree. The fitting with linear functions is called linear regression, which tries to minimize a commonly used error metric, the *mean squared error*, which is defined as the mean $\frac{1}{n}\sum_{i=1}^{n}$ of the difference between the original data points y_i and the prediction $\hat{y}_i$ [9].

$$\text{mean squared error} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{1.3}$$

As the degree of the polynomials is getting higher, the data points are fitted via nonlinear regression. More degrees in a polynomial mean more coefficients and that more data points can be fitted exactly. A polynomial of degree 4 as shown in figure 1.5a results in a very close approximation of the cosine function and would be considered a good generalized approximation of the data points, which would yield good approximation results for new data. If more data points are fitted exactly with polynomials of increasing degree, new data points of the cosine function would be fitted poorly, which is depicted in figure 1.5c.



(a) Good approximation (b) Underfitting (c) Overfitting

Figure 1.5: Samples of approximations [7].

There are a lot of methods to prevent over- and underfitting. One, the dropout by Srivastava et al. [10], is discussed in section 1.3.7. This method is used in the algorithms that have been developed for the segmentation of blood vessels.

The last point of this section is concerned with the experience E from the definition of [3], which is usually divided into three categories:

1. Supervised Learning: The machine is given different example inputs and their accompanying outputs. These outputs represent the desired solution to the input and the machine tries to predict the correct label from the given input. The classification example for the MNIST data set described above is a common reference

for a supervised learning problem. Also, the objective of this thesis, the segmentation of blood vessels, evaluates supervised learning methods. A detailed description and definition of supervised learning are given in [11].

2. Unsupervised Learning: In contrast to supervised learning, machines process unlabelled data. The objective is not the prediction of some labels, but rather to learn useful properties from the data set. Most of the tasks of unsupervised learning algorithms are occupied with finding structures in data, such that the density distribution or anomalies can be found [4]. An architecture that is widely adapted to the application of unsupervised learning are artificial neural networks, which are discussed in section 1.3.

3. Reinforcement Learning: While supervised learning provides the correct solution in the form of labelled data, reinforcement learning lets the machine, called *agent* in this context, make up a self-learned strategy. The agent interacts with its environment and depending on its state s_t can perform certain actions $a_t \in A(s_t)$. Determined by the agent's action the machine moves to a new state s_{t+1} and is rewarded or punished with $r_{t+1} \in \mathbb{R}$. The agent's objective is to maximize the rewarding function:

$$R_t = \sum_{k=0}^{T} \gamma^k \cdot r_{t+k+1} \mid 0 \leq \gamma \leq 1 \text{ [12]} \tag{1.4}$$

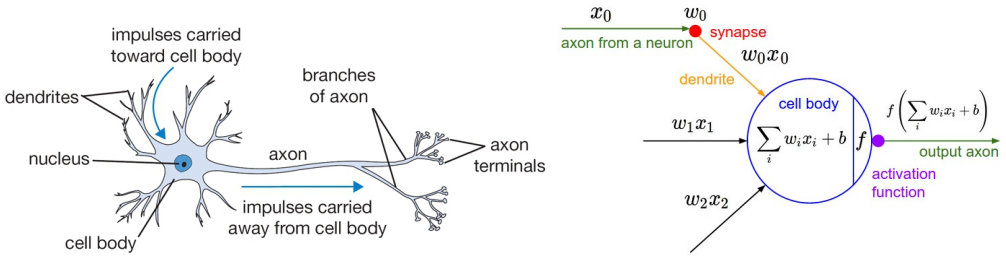
There are a lot of different machine learning approaches, such as Bayesian networks, support vector machines and many more, which are, however, not the focus of this work. This thesis focuses on a specific kind of artificial neural networks, the convolutional neural networks, which are described in the next sections.

1.3 Artificial Neural Networks

1.3.1 Biological analogue

Artificial neural networks are computational networks, which are inspired by the structure and basic function of our biological neural network. The biological neural network consists of nerve cells, the *neurons*, which can process and transmit information via electrical and chemical signals that are passed to them. These neurons are connected via *synapses*, receive signals from so-called *dendrites* and output signals along an *axon*, which is then passed to dendrites of other neurons. The dendrites pass the incoming signal to the neuron's body, where all incoming information gets summed up. The signals from other nerve cells are not equally weighted, they differ in prioritization. If the strength of the signals reaches a certain threshold, the neuron outputs a signal, called the *action potential*, along its axon, where it is transferred to synapses and dendrites of other nerve cells [4]. The schema of the biological model is shown in figure 1.6a.

The computational model, as depicted in figure 1.6b also consists of interconnected artificial neurons, which are waiting for inputs x_i from other neurons. These inputs have associated weights w_i, which represent the prioritization of incoming information and can be learned. In the artificial neuron's body, all incoming weighted information is summed up $\sum_{i=0}^{n} w_i x_i + b$ and information is passed to the next neuron, based on an *activation function* $f(\sum_{i=0}^{n} w_i x_i + b)$. b is referred to as the *bias*, which adds predefined factors to the sum of the product of weights and features. As these factors are always added to the summation of a neuron's input, it is used to steer the network to a specific decision, which biases the network from the very beginning of the training [13].



(a) Biological model of a neuron. (b) Mathematical model of a neuron.

Figure 1.6: Analogy of biological and artificial neural networks [14].

1.3.2 Architecture

For a better understanding of the basic concepts of artificial neural networks, the theory will be explained with the aid of the classification task of the MNIST data set. The depicted network does not solve the classification task optimally and a better approach is given in section 1.4. As described in the previous section, the goal is to predict the handwritten digit, which is portrayed in the input image. The output of the neural network should be a vector of size 10, representing the probabilities for each class. Optimally the network would output a single 1 for the correct label and 0 for all other labels. In case of the 5 that is depicted in figure 1.3 the output should be $(0, 0, 0, 0, 0, 1, 0, 0, 0, 0)^T$. Practically the network will output scores for each class, where the highest value represents the predicted label. The basic idea of the network's learning algorithm is to process each feature in an input image by properly adjusting weights, such that the desired outcome is achieved.

Each single pixel of the input image (28×28 pixels) is fed into the neural network, called the *input layer*, as a flattened *feature vector* $\vec{x}$ of size 784, where x_i is the i-th pixel value or feature, just as depicted in figure 1.6b. The values are multiplied with a real-valued weight, which is one of the learnable parameters and are then passed to the next neuron. The next layer of neurons is called *hidden layer*, of which multiple can exist and which, if there are plenty of them, define a deep learning network. Inside these neurons, the incoming weights are summed up and put into an activation function, of which only the most important are discussed in this thesis. For the first neuron of the hidden layer the

calculated output is: $out_0 = f(\sum_{i=0}^{n}(w_{0i0}x_i) + b)$, where w_{0i0} are the weights from all input neurons connected to the first neuron of the hidden layer, b is the bias and f the activation function. The connections from one layer to the next layer, the weights and the usage of neurons themselves are learnable parameters.

Following the example depicted in figure 1.7 the neurons of the first hidden layer are then connected to the output layer, where each neuron represents a different label. The output from the activation functions of the hidden layer is again passed to the 10 output neurons. All incoming values from the hidden layer neurons are summed up and sometimes put into an activation function again. Often the incoming values are just added because they represent the final class scores, on which the network's error is calculated. In the continuing example, the inputs for the output layer will also be put into an activation function.

The class scores for label 0 would be $f(\sum_{i=0}^{n}(w_{1i0}out_i) + b)$, where w_{1i0} are the weights from all first hidden layer neurons connected to label 0 output, out_i is the output of the *i-th* hidden neuron, f is an activation function (not necessarily the same as for the hidden layers) and b is the bias.

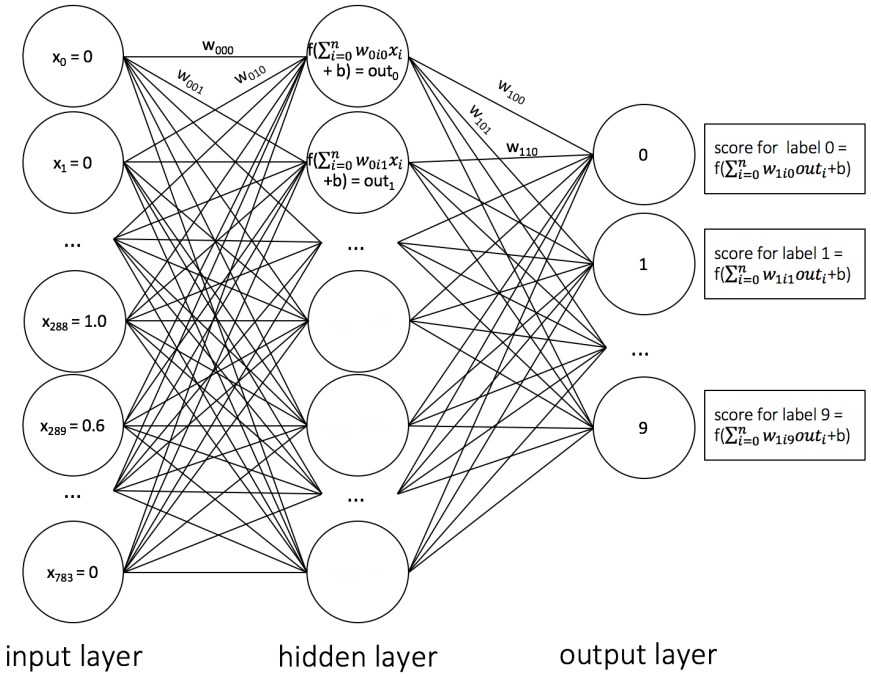


Figure 1.7: Architecture of a simple Neural Network with one hidden layer.

1.3.3 Activation functions

Before explaining the adjustment of weights, some common activation functions *sigmoid*, *ReLU*, *tanH* and *softmax* are described. An activation function, in general, is a

mathematical function, which takes a set of inputs and performs a fixed mathematical operation on it. The desired behaviour of these functions makes machine learning in neural networks possible. The idea is that small changes in the network's weights also cause only small changes in the network's output [15].

1. Sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1.5}$$

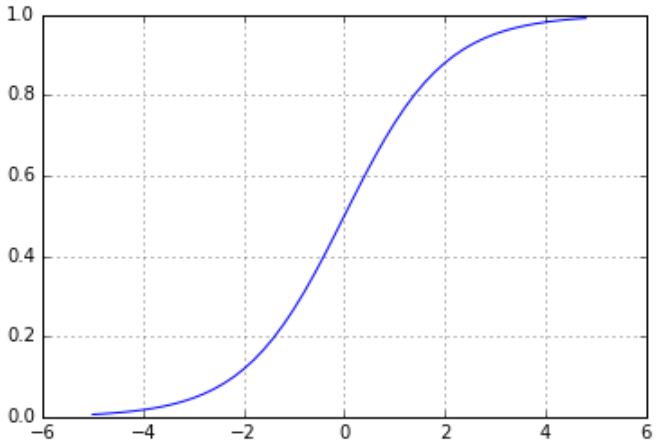


Figure 1.8: Sigmoid function.

The sigmoid function takes a real-valued input and outputs a real value between 0 and 1. With large positive numbers, the exponential term is close to 0 and the output is close 1. Large negative inputs result in an output close to 0. Small changes in the weights and the bias also only have a small effect on the output and can be even approximated as a linear function as shown in equation 1.6. The local linearity results from the smoothness of the sigmoid function, which can be approximated with a Taylor series of first-order. It makes it easy to figure out, how to change the weights and the bias to change the output [16].

$$\Delta output \approx \sum_{i=0}^{n} \frac{\partial output}{\partial w_i} \Delta w_i + \frac{\partial output}{\partial b} \Delta b, \text{ [15]} \tag{1.6}$$

A graphical representation of the sigmoid function is shown in figure 1.8.

2. ReLU (rectified linear unit) function:

$$ReLU(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} \tag{1.7}$$

The rectified linear unit function is a very simple function that only allows positive values. If the input is negative, it outputs 0. If the input is positive, the input value itself is returned. ReLU is one of the most used activation functions because it has advantages over the sigmoid function, like the acceleration of the convergence of the *stochastic gradient descent* [17], etc.

A graphical representation of the rectified linear unit function is shown in figure 1.9.

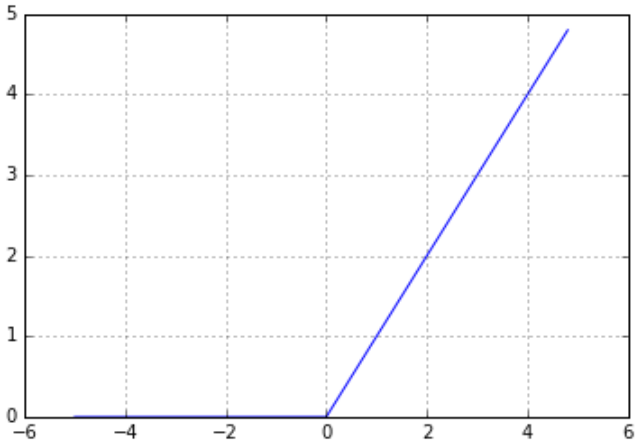


Figure 1.9: ReLU function.

3. TanH function:

$$\tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{1.8}$$

The hyperbolic tangent function takes real values and puts them in the range of -1 and 1. A graphical representation of the hyperbolic function is shown in figure 1.10.

4. Softmax function:

$$\sigma(\vec{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}} \mid i = 1, \ldots, k \tag{1.9}$$

The softmax function not only maps all elements of a vector $\vec{x}$ in the range of 0 and 1, the resulting values also add up to 1. This represents a probability distribution over k classes [18].

1.3.4 Loss functions

The network learns by adjusting its weights according to the error it has made, when predicting a label. For the estimation of the error, there exist different error estimation

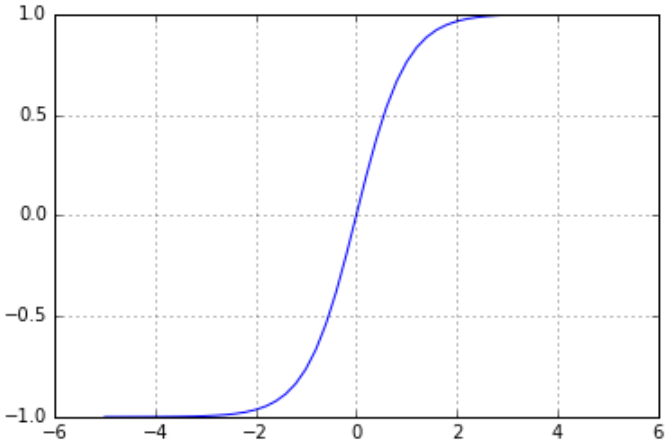


Figure 1.10: TanH function.

functions, so-called *loss functions*. Intuitively the loss will be low, if the network predicts well on the data set and high, if the network predicts poorly. Two important loss functions described in detail:

- Squared error loss: The squared error loss is another naming convention of the already introduced *mean squared error* in equation 1.3 with $\hat{y}$ being the label vector of size k, where k is the number of specified classes (in the MNIST classification task 10) and y the prediction. y_i is the *i-th* element of the label vector and $\hat{y}_i$ is respectively the *i-th* element of the prediction. The squared error loss is then defined as:

$$J(\vec{w}) = \frac{1}{k}\sum_{i=0}^{k-1} ||\hat{y}_i(\vec{x},\vec{w},\vec{b}) - y_i(\vec{x},\vec{w},\vec{b})||^2 \tag{1.10}$$

- Cross entropy loss: The cross entropy for a predicted distribution q and the wanted distribution p is generally defined as:

$$H(p,q) = -\sum_x p(x)\ln q(x) \tag{1.11}$$

  Applied to a loss function for prediction $\hat{y}_i$ and the desired output y_i:

$$J(\vec{w}) = \frac{1}{k}\sum_{i=0}^{k-1} H(p_i,q_i) = -\frac{1}{k}\sum_{i=0}^{k-1}\Big[y_i\ln(\hat{y}_i) + (1-y_i)\ln(1-\hat{y}_i)\Big] \text{ [19]} \tag{1.12}$$

  The cross entropy loss is often used together with the softmax activation function from equation 1.9 [18].

Assuming the initial weights are randomly assigned and the network is fed with a handwritten 5 of the data set. If the output layer receives a vector $\vec{v}$ =[0.0309, 0.0429, 0.0309, 0.732, 0.009, 0.09, 0.0129, 0.0549, 0.0219, 0.0219, 0.0829] for example, before applying the sigmoid activation function, the result after the sigmoid function would be $\hat{y}$ = [0.5077, 0.5107, 0.5077, 0.6752, 0.5022, 0.5224, 0.5032, 0.5137, 0.5054, 0.5054, 0.5207] and would predict label 3, because 0.6752 is the highest value of the prediction. As the output should be $y_i = [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]$, the mean squared error loss is approximately 0.2738.

If we apply the softmax function to $\vec{v}$, we get a prediction of $\hat{y}$ = [0.083, 0.084, 0.083, 0.166, 0.081, 0.088, 0.081, 0.085, 0.082, 0.082, 0.087] and the cross entropy loss would be 0.3084.

A neural network learns by minimizing the loss function because then the error between prediction and actual value gets smaller and smaller. Some more examples of loss functions, which are used in praxis and a comparison of the results these functions yield, are given in [16].

1.3.5 Gradient descent

The loss function is dependent on the input features, the weights, and the bias. Because the input features are fixed values, which can only be properly preprocessed as discussed in chapter 2, the network must adjust the weights and the bias. For a very simple neural network, for example the one depicted in figure 1.7, there are 784(number of input values)×784(number of hidden neurons) + 784(number of hidden neurons)×10(number of output neurons) + 784(number of biases in the hidden layer) + 10 (number of biases in the output layer)= 623290 weights to be adjusted. Due to the nature of the matrix multiplication, the bias vector could be folded into the weight matrix, which would give us a single parameter matrix for optimization [17]. For reasons of simplicity, only the weight matrix will be used for minimizing the loss function.

The enormous number of weights makes it computational impossible to get good results with a brute force method by decreasing and increasing each single weight and then measuring the effect on the loss function. We face a complex optimization problem.

A good minimization algorithm for the loss function, is the so-called *gradient descent algorithm*. It minimizes the output of functions iteratively by starting with an initial set of parameters and continuously moving towards a set of parameters, which minimizes the loss. It adjusts the parameters by calculating the gradient of the function, i.e. the partial derivatives in the direction of the function's parameters, mathematically formulated:

$$grad(f) = \nabla f = \frac{\partial f}{\partial x_i}\hat{e_1} + \cdots + \frac{\partial f}{\partial x_n}\hat{e_n} \mid \hat{e_i} = \text{unit vector, [20]} \tag{1.13}$$

To keep it simple, a short introduction to the gradient descent algorithm with 2 weights and the objective of minimizing the squared loss function will be given. Assuming we

want to predict how good a student can perform in terms of percentage y, based on the hours of sleep x_1 and the hours of studying x_2. A basic table would look like this:

Table 1.1: Example inputs

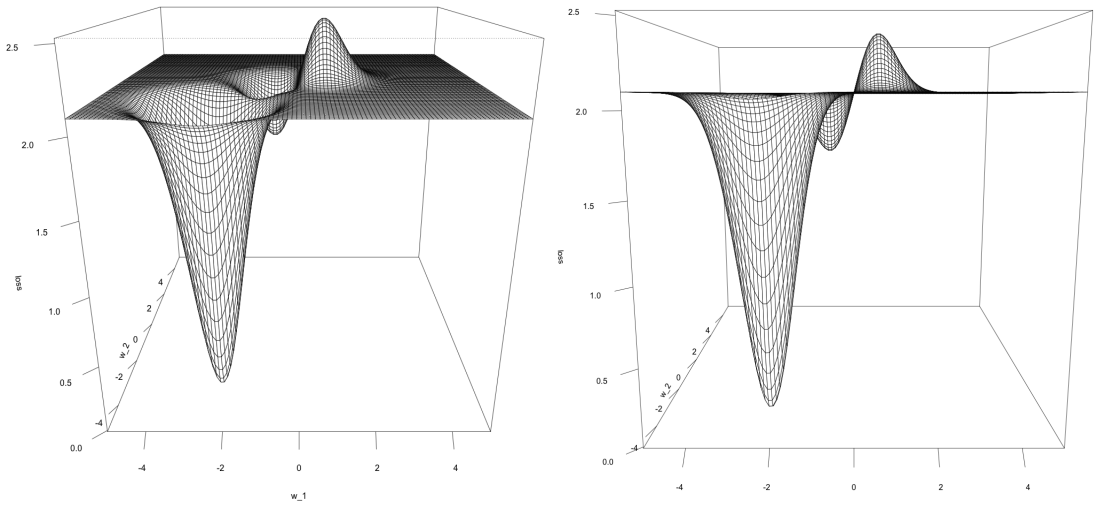
Hours of sleep	**Hours spent studying**	**Test score**
8	30	95
7	25	89
5	10	45
6	15	56
9	10	50
4	30	67
...	...	...

The inputs x_1 and x_2 can be weighted with real-valued weights w_1 and w_2 to approximate the output y. The prediction $\hat{y}$ for the *i-th* element of x_1 and x_2 is $\hat{y}_i = x_1^{(i)} w_1 + x_2^{(i)} w_2$. The squared error loss $J(w_1, w_2) = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$ with n being the number of samples that are used for training. The goal is to minimize this loss function of two parameters. The gradient descent algorithm starts with an initial guess of the weights w_1, w_2 and then changes the weights according to:

$$w_i \coloneqq w_i - \alpha \frac{\partial}{\partial w_i} J(w_1, w_2) \mid i = \{0, 1\} \text{ [20]} \tag{1.14}$$

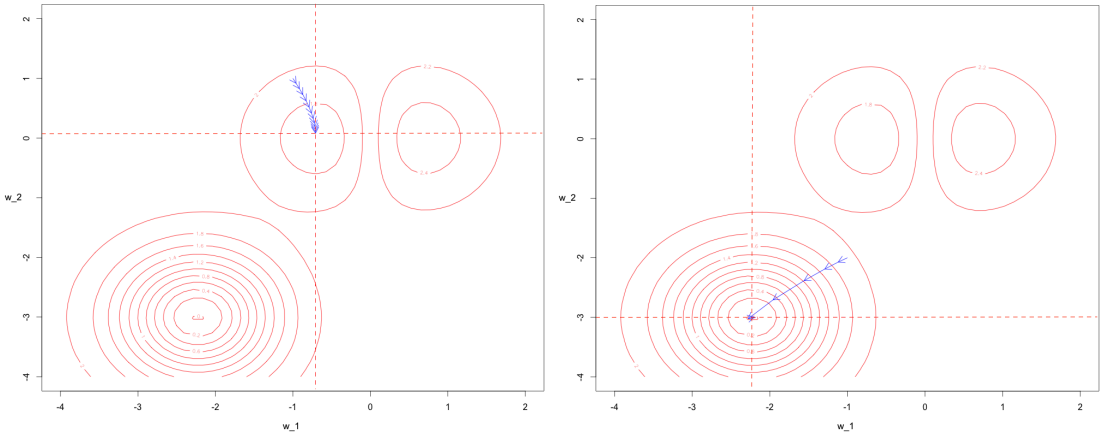
This results in adjusting the weights according to the steepest improvement for all weights. α is the step-size, also called the *learning rate* and is usually of the order of 10^{-1} down to 10^{-7}. The algorithm adjusts the weights until it converges, hopefully in a good minimum of the loss function. If α is too small, the gradient descents very slowly and maybe does not converge at all. If α is too big, a good minimum could be stepped over. Also the initialization is important, because the algorithm can only locally calculate the gradient and of course the loss function is essential. If the loss was approximated by $J(w_1, w_2) = 2.1 + w_1 e^{-(w_1^2 + w_2^2)} + w_1 e^{-((w_1+2)^2 + (w_2+3)^2)}$ for example, as it is depicted in figure 1.11, there are two distinct minima, where the gradient descent algorithm can be caught, depending on the initialization. If the algorithm starts for instance at $(w_1, w_2) = (-1, 1)$, the gradient gets caught in the local minimum as depicted in figure 1.12a, which stops after 14 iteration at approximately $(w_1, w_2) = (-0.75, 0.1)$ and an approximate loss of $f(w_1, w_2) = 1.68$. If the algorithm starts at $(w_1, w_2) = (-1, -2)$ as depicted in figure 1.12b it stops after 6 iterations at the minimum in the range of $w_i \in \{-5, \ldots, 5\}$.

The squared error loss function described in equation 1.10 has a much simpler gradient and is used because the gradient descends to good results and is not easily caught in bad local minima.



(a) Loss function plot. (b) Loss function plot (front).

Figure 1.11: An example loss function with two distinct minima.



(a) Gradient descent at $(w_1, w_2) = (-1, 1)$. (b) Gradient descent at $(w_1, w_2) = (-1, -2)$.

Figure 1.12: Gradient descent based on initialization.

1.3.6 Stochastic gradient descent

The loss function from above is only dependent on two weights and even bigger training sets can be calculated quickly. In a big neural network, there are, however, millions of weights and thousands of samples. If the gradient would be calculated for the whole training set, the computation would take relatively long for a single improvement step. The stochastic gradient descent approximates the gradient descent algorithm stochastically, by taking one training sample or a small subset of the training set, calculating the gradient on this sample and then updating the weights according to this gradient. Progress is

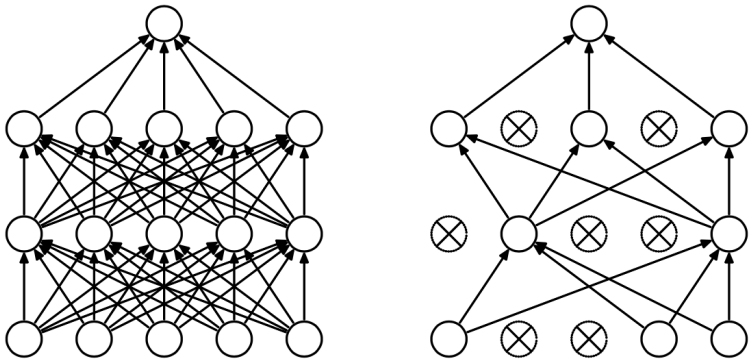
made right away and helps the algorithm converge faster. However, as the updates are done right away, the minimum may never be reached and the algorithm oscillates around the minimum of $J(\vec{w})$. Different stochastic gradient descent variants and their respective results are shown by L. Bottou in [21].

1.3.7 Regularization

As introduced in the section about machine learning 1.2, one of the core problems of machine learning algorithms is overfitting. To prevent overfitting the amount of data can be increased, the network can be made smaller or the network can be explicitly designed to reduce the test error, which is called regularization [4].

There are many different variants of regularization, of which some are well described by Goodfellow et al. [4], like the L^2 parameter regularization. I will only describe one technique, which was developed by Srivastava et al. [10] in the year 2014, called *Dropout*.

Dropout modifies the network itself by reducing the number of active neurons. The Dropout mechanism temporarily removes a neuron, either hidden or visible, with an independent probability p. If p is set to 0.3 the network is thinned to approximately 70% of its original neurons. An example is shown in figure 1.13. The technique yielded very good results, of which some are shown in the work of Srivastava et al. [10].



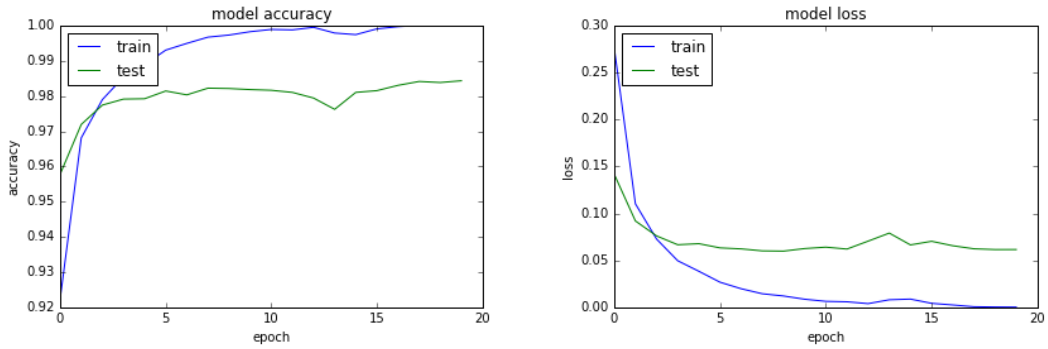
(a) Standard Neural Net (b) After applying dropout.

Figure 1.13: Visualization of Dropout by Srivastava et al. [10].

1.3.8 Outlook

The example from the previous sections is a so-called *feedforward neural network* because information only flows from a previous layer to the next layer. There are other architectures, where information can be passed in different ways, such as in *recurrent neural networks*, where information can be passed backwards in form of a *feedback loop*. As the algorithms implemented for this thesis are only feedforward convolutional neural networks, other networks will not be discussed.

This simple neural network, with only one hidden layer, ReLU activation function for the hidden layer, softmax activation function for the output layer, cross entropy for the loss function and an *optimizer* that uses stochastic gradient descent, yields an error rate of 1.55%, i.e. an accuracy of 98.45% on the MNIST data set with a self-written program in Keras, an open source library for neural networks. The training time is very low with only 2.5 minutes. Figure 1.14 shows the training history with typical accuracy and loss curves. Slight overfitting on the training data occurred, which can be interpreted from the gap between the training and test curves.



(a) Accuracy of the model.

(b) Loss of the model.

Figure 1.14: The example network's training history on the MNIST data set.

The network yields good results on an easy example because the weights can be properly adjusted. However, if the objects to be classified get more complex and especially if they get shifted, rotated, etc., such a simple neural network will not be sufficient.

The combination of possibilities seems to be endless and the tuning of hyperparameters for good results, like adjusting the size of the network, selecting an activation function, choosing correct preprocessing, a good loss function, etc. is often achieved by trial and error.

1.4 Convolutional Neural Networks

1.4.1 Architecture

A Convolutional Neural Network (CNN) is a specific kind of a feed-forward artificial neural network, that is known to be translation and rotation invariant, meaning that the outcome of the network does not change, if the processed objects are translated or rotated. The architecture consists of an input, an output and multiple hidden layers with nonlinear activation functions. The hidden layers are either so-called *convolutional* layers, *pooling* layers and *fully connected* layers, which are just connecting all neurons of the current layer to all neurons of the next layer. The typical structure of a CNN is built

up from a sequence of convolution layers and pooling layers, followed by one or multiple fully connected layers, which are connected to the output layer [20].

The neurons of the network are structured differently from a classical neural network because they are often used for a learning task on images. In the following, CNNs with respect to classification of objects in images will be described in more detail, because this is the thesis' main approach.

The neurons are structured in a 2D or 3D order, i.e. the width, height (and depth) of the network's input. Images of the first mice embryo data set, described in section 1.1, have a width of 2279, a height of 1753 and a depth of 1 because the images are grayscale and only have one colour channel. RGB images, for instance, would have a depth of 3, one slice for every colour channel. If each layer would use only fully connected layers, it would result in a very fast growing number of weights, resulting in slow computation. It is, however, possible, to only connect to a subset of neurons from the previous layer, due to *local connectivity* described in section 1.4.4. The size of this subset is called the *receptive field* and is a hyperparameter, that has to be tuned.

The first convolutional neural network was developed by Yann LeCun in 1998 in his work *Gradient Based Learning Applied to Document Recognition* [20] with an architecture known as the *LeNet-5*, which is depicted in figure 1.15 and was used for classifying handwritten digits on images. The architecture transforms the pixel values of the input images layer by layer to the final class scores.

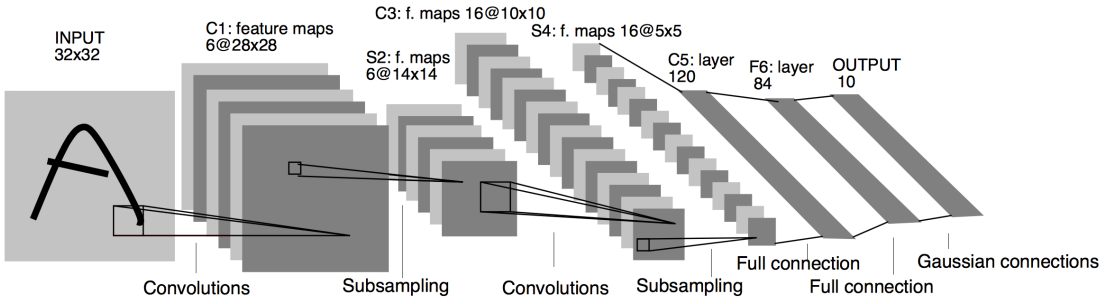


Figure 1.15: LeNet-5 architecture [20].

1.4.2 Convolution

Convolution itself is a mathematical operation (denoted with $*$) on two functions, which produces a third function, generally defined as:

$$(f * g)(n) := \int f(k)g(n-k)dk \text{ [4]} \tag{1.15}$$

In terms of convolutional neural networks the function f is referred to as the *input*, the function g as the *kernel* and the output as the *feature map*. The discrete convolution for an image I and a kernel K is defined as:

$$(I * K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \text{ [4]} \tag{1.16}$$

With respect to the MNIST example, the input layer neurons are structured as a $28 \times 28 \times 1$ matrix, because the images have a width of 28 pixels, a height of 28 pixels and are grayscale. Intuitively the kernels are a set of digital image filters, which are moved over the images and the feature map is the result of the filtered images. The filters are used to highlight certain features of an image, such as edges and can be learned throughout the training of the network. The filtering can be understood as sliding the filter across the width and height of the input image, which computes the dot product of the image pixel values and the filter values [22] and is depicted in figure 1.16. The resulting feature map is a 2D array, which shows higher values in those areas, where specific features (according to the filter) have been detected.

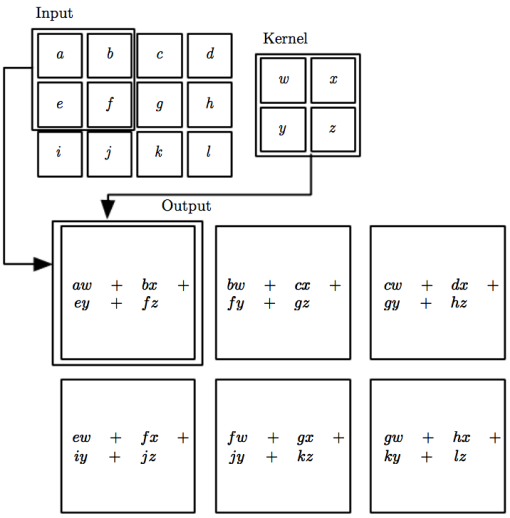


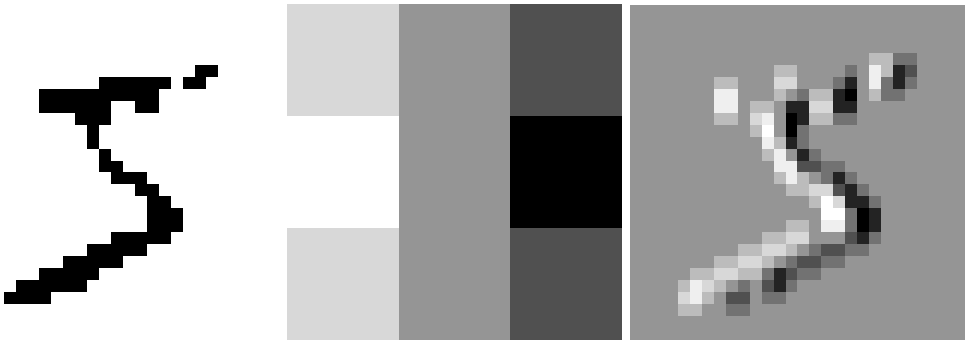
Figure 1.16: Convolution with a 2x2 filter [4].

A possible 3x3x1 filter, a version of the vertical *Sobel-operator* or *Sobel-filter*, which is used for vertical edge detection [23], is defined as:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

The kernel always has the same depth dimension as the input image, which is 1 in a grayscale image or 3 in an RGB image. The width and height of the kernel is a hyperparameter, just as the *stride* with which the kernel is moved along the image and which affects the height and the width of the feature map. If the stride is 1 the kernel is moved to the next pixel. If the stride is 2 the kernel is moved to the next but one, etc. Another hyperparameter that has to be tuned is the number of kernels used, which affects the depth of the feature map stack.

Convolving figure 1.17a, which shows an image of the MNIST data set, where only 0 and 1 as values were used, with the Sobel-filter from figure 1.17b with stride 1 results in the feature map depicted in figure 1.17c.



(a) 5 from MNIST. (b) Sobel-filter (c) Sobel-filtered 5.

Figure 1.17: Example of a Sobel-filtered image.

Intuitively the network will learn the kernels that will activate, i.e. that will have relatively high values, when filtering the input. The network uses different kernels to detect different features in an image. If the network takes $28 \times 28 \times 1$ images as an input and uses 16 different $5 \times 5 \times 1$ kernels with stride 1, it would result in 16 different 28×28 feature maps.

1.4.3 Pooling

The feature maps of the convolution layer are run through nonlinear activation functions, described in section 1.3.3 and are subsequently processed in *pooling layers*, a form of nonlinear down sampling. The only pooling discussed in this thesis is the so-called *max-pooling*, which is commonly used in convolutional neural networks. Krizhevsky et al. [24] define the max-pooling layer as the "maximum activation over non-overlapping

rectangular regions of size (Kx, Ky)", which results in position invariance and downsamples 2D images by a factor of K_x and K_y. The max-pooling takes the maximum of the $K_x \times K_y$ values and discards the rest, resulting in a downsampling of the input. A 2×2 max-pooling would downsample a 28×28 pixel feature map to a 14×14 pixel feature map and therefore discard 75% of the activation features. An example of a 2×2 max-pooling is shown in figure 1.18.

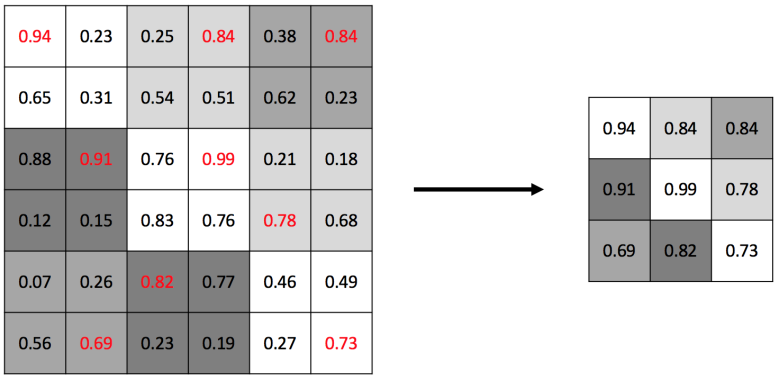
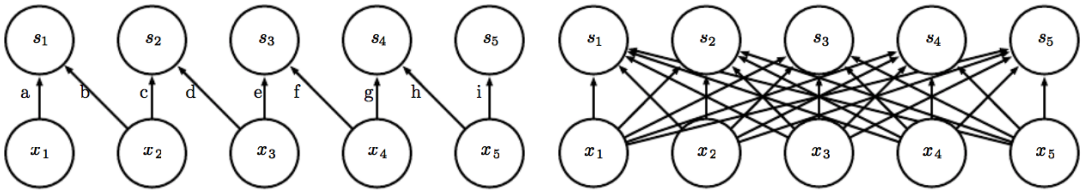


Figure 1.18: 2x2 max-pooling.

Intuitively, the relative location compared to other features is more important than the exact location of that feature. The pooling prevents overfitting to some extent, makes the features translation invariant and leads to faster convergence by selecting superior invariant features [24].

1.4.4 Local connectivity

Instead of connecting to all neurons of the previous layer, CNN architectures connect only to a local region of the previous layer. This is especially used with images because the network has to learn local structures, which is achieved by letting neurons only receive local information.



(a) Locally connected neurons [4] (b) Fully connected neurons

Figure 1.19: Locally and fully connected neurons.

1.4.5 Parameter sharing

The number of parameters in a convolutional neural network can get very large, especially for a deep convolutional neural network with many layers would result in a very slow learning process. To control the number of parameters, the neurons share weights and biases. The intuition is, that if a feature at a certain position helped to minimize the loss function, this feature will also be helpful at another position [4]. With respect to this thesis' objective, the detection of blood vessels in images, circular features are important. If the network would not share weights, it would have to learn to detect these circular structures at every image location. Thus, these circular structures would have to be present at every location in the image.

CHAPTER 2

Methods

As already mentioned, this thesis' practical objective was to implement and assess different algorithmic approaches for detecting blood vessels in given biomedical images. This is a classification or segmentation task in the field of supervised learning.

In section 2.1 the hardware and software, which were used for implementing the algorithmic approaches, are described for reasons of accountability. In section 2.2 the preprocessing, the implemented architecture and the project setup for the segmentation task in two dimensions are described. Section 2.3 describes the same points for the segmentation in three dimensions. Chapter 3 contains the results of the methods from this chapter and some recommendations for further experiments.

2.1 Technology

The networks from section 2.2.2 and section 2.3.2 have been developed with Python v3.4.5. and were built up with Keras v1.2.2 running on Theano v0.9.0, a library for numerical computation. Also, NumPy v1.11.1 and SciPy v0.18.1 for scientific programming were used extensively for image processing. The hardware consists of an Intel Core i7 CPU with 12GB RAM and a local NVIDIA GeForce GTX 285 GPU with 1GB GDD3 memory. The training of the networks was run on external NVIDIA GeForce Titan X GPUs with 12GB G5X5X memory, 1531 MHz boost clock and a memory speed of 10 Gbps.

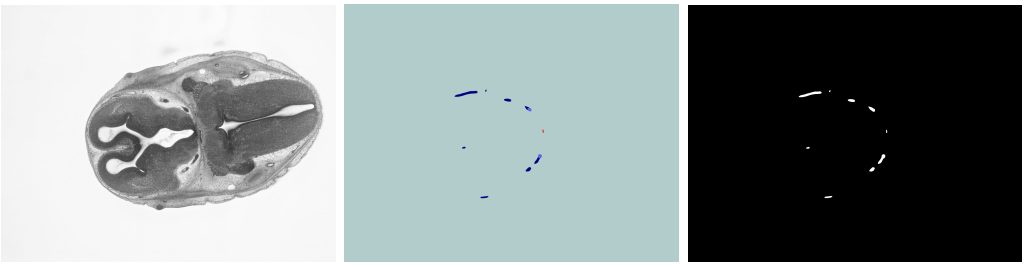
2.2 Segmentation in 2D

Segmentation in 2D means classifying vessels in slices of the HREM image stack. The network is fed with 2D images and returns 2D slices, where the vessels of that slice are segmented. The predictions can be stacked together to return the final volume.

2.2.1 Preprocessing

The data from the mice vessel embryo study does not only contain the HREM images of the mice embryos but also accompanying labels. Figure 2.1a shows a 2D slice of the study and the accompanying labelled slice in figure 2.1b , where vessels have been segmented and where different vessels have been coloured differently.

The labelled slices were transformed to black and white masks only containing zeros and ones. $(x_i, y_i) = 0$ depicts, that there is no vessel at (x_i, y_i) and $(x_i, y_i) = 1$ depicts, that there is a vessel at this location. These black and white masks were used as so-called *ground truth masks* as the network's objective, i.e. what should be predicted and is used to measure the network's error rate. An example is depicted in figure 2.1c.



(a) Original HREM slice (b) Labelled HREM slice (c) Ground truth mask

Figure 2.1: Mice embryo sample.

The data has to be preprocessed before it is fed into the network. First of all, the whole image is too big for the learning task, because a single slice contains 2279×1753 pixels with a mean representation of 0.07% vessel tissue in those areas, where vessels are actually present. There are also a lot of slices without any representation of vessels, as shown in figure 1.1. Furthermore, some images are noisy and pixel values are not centered around the origin, which will be described shortly. I have mainly used five distinct preprocessing approaches:

- Random patch extraction: Ciresan et al. [25] used to train on local regions (patches) around pixels, which provided a lot more training input than training on the data set itself. I have only used quadratic patches of different sizes (64×64, 128×128, 192×192, 256×256 and 512×512), of which the results are discussed in chapter 3.

  I have used two different mechanisms for the random patch extraction. Both mechanisms extract patches randomly around pixels, where there is a vessel present and extract the accompanying masks. Therefore all extracted patches contain vessels at a random place/at random locations in the patch. This procedure was chosen because I wanted to experiment if the network can learn the vessel's tubular structures in general. The patch size, i.e. height and width of a patch, along with the number of patches are hyperparameters. Different approaches have been

used and the results are described in chapter 3. The general idea of the 2D patch extraction is depicted in figure 2.2.

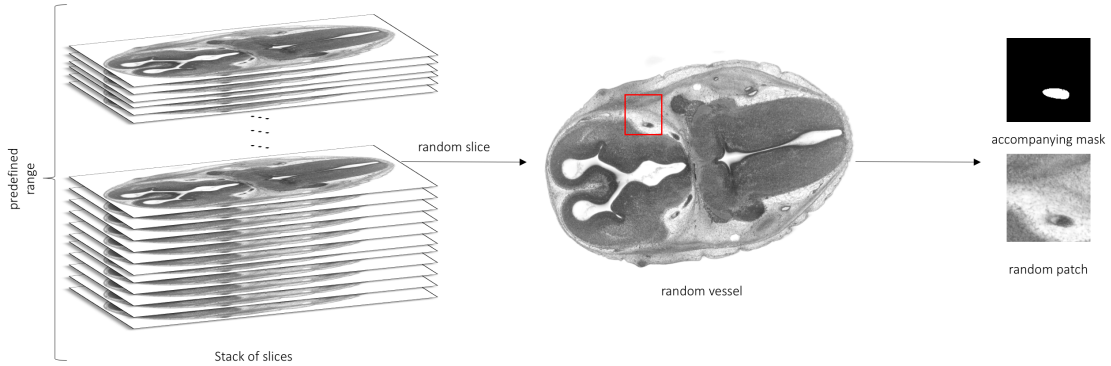


Figure 2.2: Random patch extraction.

One mechanism uses so-called *generators*, which are Python functions that create random patches dynamically in a given range. The generator is fed with slices of the original image and returns a random patch or a stack of random patches. The other mechanism returns a static stack of patches of predefined size.

The network is fed with some thousand patches, which is discussed in more detail in chapter 3. The big advantage of the generator is concerning memory and computation time because the generator function does not have to save a large stack of patches due to the dynamic generation during training.

- Zero-mean unit variance normalization: I have used patch-wise normalization, which results from subtracting the mean from every patch and subsequently dividing the patches by their respective standard deviation. The subtraction of the patch's mean results in centered data around the origin and of course a mean of 0. The division by the standard variance results in scaled data, where each patch has a standard deviation of 1. A graphical representation of the procedures is shown in figure 2.3.

- Downsampling: In later experiments, I have used downsampling. The images are downsampled by a factor of two by only taking every second pixel value and discarding the rest. This resulted in more context information with a consistent patch size and helped to denoise the data a little bit.

- Denoising: Many patches are noisy, which was troublesome for the convolutional neural network architectures because a lot of noise was classified as tubular structures, which resulted in many false-positive predictions. I have mainly experimented with the *median filter*, which takes the median of a specified window [27], for example, a 3×3 window, and is moved over every pixel of the image. The result of such a median filter is depicted in figure 2.4.

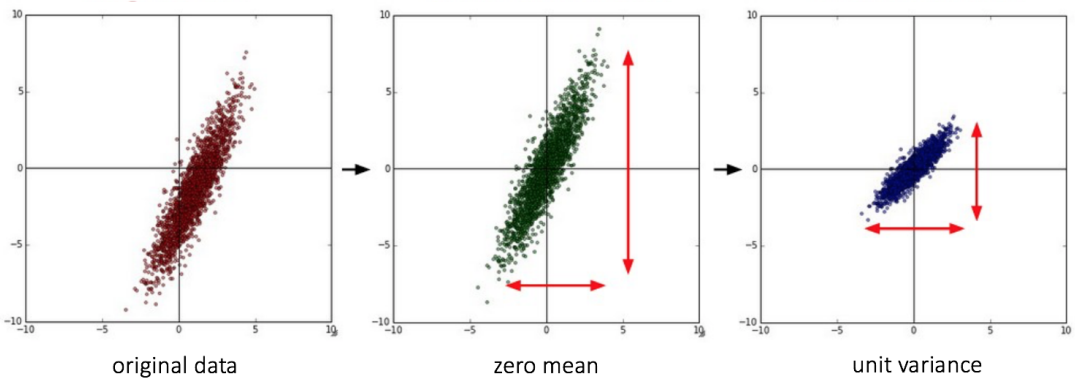
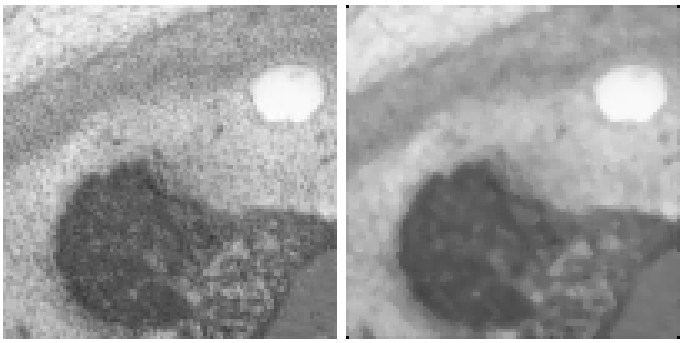


Figure 2.3: Normalization of data [26].



(a) Original Patch. (b) Filtered patch

Figure 2.4: Median filter applied to a noisy image.

- Big vessels: The generators I have built for random extraction have parameters, which specify, how much of the patch has to be vessel tissue. The reason for this is the noisiness of the data. If a vessel's diameter is relatively small and the network has to learn these small features, the network also starts to predict noise as vessel tissue. As shown in chapter 3, I have experimented with discarding those patches, that have relatively small vessels.

2.2.2 2D U-Net

I have used the 2015 *U-Net* by Olaf Ronneberger et al. [28] as the primary deep learning algorithm. The network was developed for biomedical image segmentation and has outperformed the network of Ciresan et al. [25], which has won several competitions.

The architecture is a so-called *encoder-decoder* architecture based on convolutional neural networks. The encoding part of the network reduces the height and width of the images step-by-step by sequential application of convolutions and poolings. The decoding part of the network recovers the original dimensions and preserves the object's details, such

that the network can output a segmentation map of the input image, as it is shown in figure 2.1c. The objective of my algorithm is to output a map, where 1 depicts vessels and 0 everything else, such that the vessels are segmented from the rest of the image.

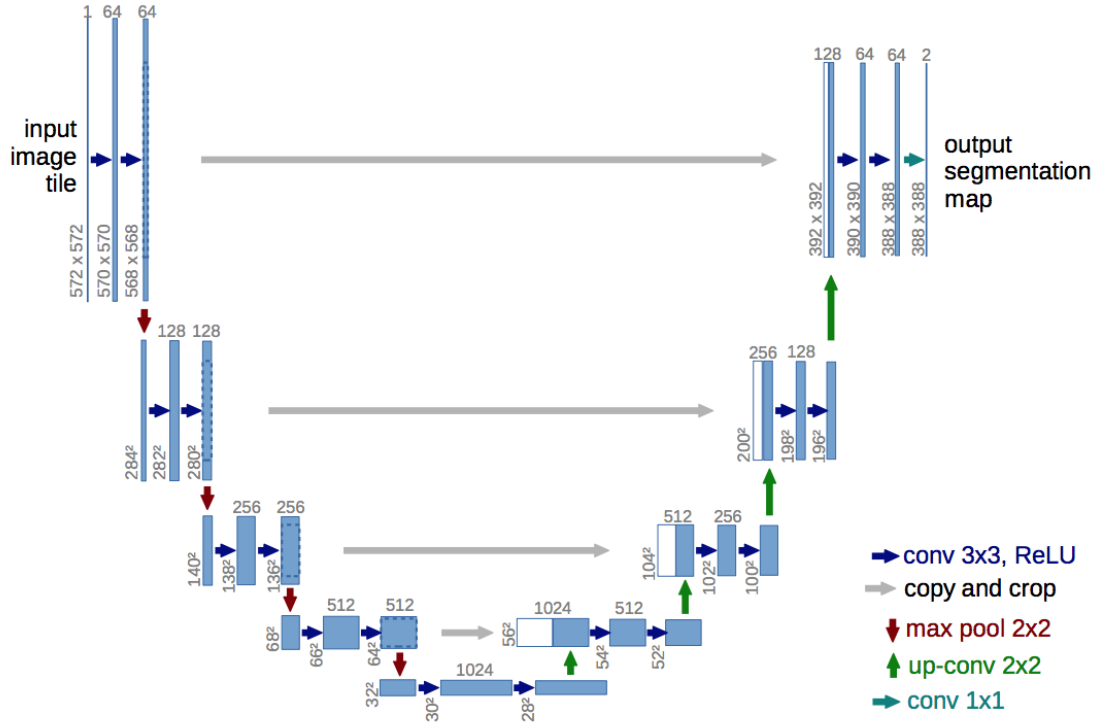


Figure 2.5: U-Net 2D [28].

The original architecture is shown in figure 2.5. The input consists of 572×572 pixel images. The network starts encoding the data on the left side with 64 3x3 convolution filters, which are followed by rectified linear unit activation functions, defined in equation 1.7. Each convolution layer crops 2 pixels of the border. After convolution, the feature maps are pooled by a 2×2 max pooling operation with stride 2, which reduces the height and width of the image by a factor of 2. Then again follows convolution, ReLU activation, and pooling until the input is flattened to 1024 30×30 feature maps. There are no fully connected layers in this architecture.

Now the decoding part starts from the 1024 feature maps with corresponding up-convolutions, which halve the feature channels. The up-convolutions are concatenated with the feature maps from the same "layer" of the encoding path to regain the object's details. The concatenated feature maps are then again convoluted by 3×3 filters and a ReLU activation function. This procedure is repeated until the number of up-convolution matches the number of max-poolings. The final layer uses a 1×1 convolution with a softmax activation function. This maps each of the feature vectors of the final layer to the number of output classes, which is 2 in the case of this thesis' objective. Due

to the cropping of borders and the halving from the encoding part of the network, the output segmentation map has less width and height than the original input in the original architecture. The network uses the cross entropy loss function defined in equation 1.12.

2.2.3 Project Set-Up

The first step of the program I have developed is saving a range of the original embryo slices, converting the accompanying labelled images to ground truth masks and storing them as NumPy arrays. The embryo data has nearly no vessels in the first few hundred and the last few hundred slices. Therefore I have selected between 400 and 1000 slices from the initial volume. The slicing started from a specified index, which ranged between index 600 and 1000, where most of the vessels were located.

The stack is then passed to the self-written patch extraction functions, where the preprocessing takes place, as it is described in section 2.2.1. Some of the parameters are hyperparameters, which have to be tuned, like the size of the patches, the downsampling-rate, the functions for denoising, etc.

After the patches were extracted or the generator with the ability to extract patches dynamically was initialized, the patches are checked for consistency, like ranges and unique values of the masks, zero-mean and unit variance of the patches, if the masks are also matching the inputs, etc. For each experiment, the program creates new test folders, where the model parameters, some patches, and masks, test images, the accuracy of the model, etc. are stored.

After preprocessing and consistency checks, the program builds the U-Net architecture with Keras. A code snippet of the implementation with commentaries for the parameters is shown in listing 2.1. The input consists of the number of channels of the input images, the height, and width of the patches, the number of convolution filters that should be used in every layer, the activation functions after the convolutions, the activation function of the final layer, the learning rate of the optimizer *Adam* [29] and the dropout rate.

For activation functions, after the convolutions, I have used the rectified linear unit function defined in equation 1.7. For the activation function of the final layer, I have used a slight alteration of the softmax activation function, which is defined in equation 1.9. I have computed a pixel-wise softmax function, which takes the exponential function of the specified pixel-value and sums over all pixel-values of the input.

For the loss function, I have used a pixel-wise cross entropy and the weighted version of it, which takes the mean value of the ground truth mask and weights the occurrence of vessels stronger. The metrics on which the accuracy of the model is evaluated is called the *dice similarity coefficient* or *Sørensen index*, which is a measurement of similarity of two samples [30]. The dice definition for (binary) masks X and Y is given by:

$$dice := \frac{2|X \cap Y|}{|X| + |Y|} \tag{2.1}$$

and was adapted for the predictions and the ground truth masks of the patches. In the following, I will refer to the dice coefficient for overlapping vessel tissue as $dice_1$ and to the dice coefficient for overlapping background or non-vessel tissue as $dice_0$. For Dropout I have used values ranging from 0.2 to 0.3, discarding $20 - 30\%$ of the neurons.

```
## n_ch - number of channels of the input images
## patch_height - height of the patches
## patch_width - width of the patches
## conv - array specifying the number of convolution filters
## activ - activation_function after every convolution
## core_activation_function -  activation_function of the final layer
## learning_rate - learning_rate of the optimizer Adam
## drop_out -  percentage of weights to be dropped
## loss_function -  the loss_function for evaluating the results
def unet(n_ch,patch_height,patch_width, conv=[64,128,256,512,1024], activ,
    core_activation_function, learning_rate, drop_out, loss_function):
    inputs = Input((n_ch, patch_height, patch_width))

    #Convolution down (left side of the UNet)
    conv1 = Convolution2D(conv[0], 3, 3, activ, border_mode='same')(inputs)
    conv1 = Dropout(drop_out)(conv1)
    conv1 = Convolution2D(conv[0], 3, 3, activ, border_mode='same')(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)

    conv2 = Convolution2D(conv[1], 3, 3, activ, border_mode='same')(pool1)
    conv2 = Dropout(drop_out)(conv2)
    conv2 = Convolution2D(conv[1], 3, 3, activ, border_mode='same')(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)

    conv3 = Convolution2D(conv[2], 3, 3, activ, border_mode='same')(pool2)
    conv3 = Dropout(drop_out)(conv3)
    conv3 = Convolution2D(conv[2], 3, 3, activ, border_mode='same')(conv3)
    pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)

    conv4 = Convolution2D(conv[3], 3, 3, activ, border_mode='same')(pool3)
    conv4 = Dropout(drop_out)(conv4)
    conv4 = Convolution2D(conv[3], 3, 3, activ, border_mode='same')(conv4)
    pool4 = MaxPooling2D(pool_size=(2, 2))(conv4)

    conv5 = Convolution2D(conv[4], 3, 3, activ, border_mode='same')(pool4)
    conv5 = Dropout(drop_out)(conv5)
    conv5 = Convolution2D(conv[4], 3, 3, activ, border_mode='same')(conv5)

    #convolution up (right side of the UNet)
    up1 = merge([UpSampling2D(size=(2, 2))(conv5), conv4], mode='concat',
    concat_axis=1)
    conv6 = Convolution2D(conv[3], 3, 3, activ, border_mode='same')(up1)
    conv6 = Dropout(drop_out)(conv6)
    conv6 = Convolution2D(conv[3], 3, 3, activ, border_mode='same')(conv6)

    up2 = merge([UpSampling2D(size=(2, 2))(conv6), conv3], mode='concat',
    concat_axis=1)
    conv7 = Convolution2D(conv[2], 3, 3, activ, border_mode='same')(up2)
```

```
    conv7 = Dropout(drop_out)(conv7)
    conv7 = Convolution2D(conv[2], 3, 3, activ, border_mode='same')(conv7)

    up3 = merge([UpSampling2D(size=(2, 2))(conv7), conv2], mode='concat',
    concat_axis=1)
    conv8 = Convolution2D(conv[1], 3, 3, activ, border_mode='same')(up3)
    conv8 = Dropout(drop_out)(conv8)
    conv8 = Convolution2D(conv[1], 3, 3, activ, border_mode='same')(conv8)

    up4 = merge([UpSampling2D(size=(2, 2))(conv8), conv1], mode='concat',
    concat_axis=1)
    conv9 = Convolution2D(conv[0], 3, 3, activ, border_mode='same')(up4)
    conv9 = Dropout(drop_out)(conv9)
    conv9 = Convolution2D(conv[0], 3, 3, activ, border_mode='same')(conv9)

    conv10 = Convolution2D(2, 1, 1, activ, border_mode='same')(conv9)

    conv11 = core.Activation(core_activation_function)(conv10)

    model = Model(input=inputs, output=conv11)
    model.summary()

    model.compile(optimizer=Adam(lr=learning_rate), loss=loss_function,
    metrics=[d1])

    return model
```

Listing 2.1: Code snippet of the U-Net implementation in Keras

As soon as this model is initialized with proper parameters, the Keras command *model.summary()* from the code-snippet in listing 2.1, returns the input and output shape, the number of parameters and some other meta information of the model. With the default model from Ronneberger et al. [28], the model has 31377858 trainable parameters, which yielded the best results for my experiments, as shown in chapter 3.

The commands *model.fit()* or *model.fit_generators()*, when using generators for patch extraction, start the training with the Keras library. I have used model checkpoints, which save the best combination of weights according to the $dice_1$ score on the training set. The training commands contain additional hyperparameters, which have to be tuned, like the so-called *batch size*. The batch-size is the number of samples, which are going to be propagated through the network until an update of the weights according to the gradient descent algorithm occurs. I have experimented with batch-sizes ranging from 1, 2, 5, 10, 20, 64, 100 up to 200. The results are described in chapter 3.

During training, Keras provides feedback of the current status of accuracy and loss both on the training and the test set. The test set during training is selected by splitting the data in a certain ratio, where I have mainly used a split of validation:training of 1:10 or 1:5. Keras provides a *history()* command, which shows loss and accuracy according to the predefined metrics in a plot.

After the model has finished training, the last configuration of weights is saved as well and the test phase begins. I am extracting patches from regions the training has not yet seen and use the same preprocessing that was used in the training. The accuracy and the loss of the trained model are evaluated over the test patches and represent the ability of the network to generalize to unseen data. Afterwards, I am letting the network predict on the test and training patches and compare the results to the ground truth masks, which is shown in chapter 3.

To predict a whole image of the embryo data set, I am dividing the HREM slices into patches of the same size as the training patches. As the patches I have extracted were squares and the slices' size is not a multiple of the patches' size, I had to add some border to the slices to ensure, that I can segment the whole image. This procedure is shown in figure 2.6, where I have used 64x64 pixel patches and downsampled all slices by a factor of 2, resulting in an 1140×877 pixel slice for the embryo study with original dimensions of 2279×1753.

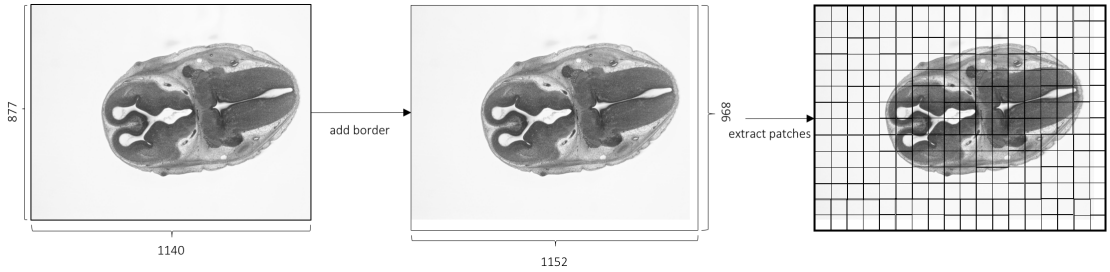


Figure 2.6: Patch extraction for a whole slice.

2.3 Segmentation in 3D

Segmentation in 3D classifies vessels in volumes of the HREM images, presenting an additional dimension for context information. The networks I have implemented have a cubic 3D input and output.

2.3.1 Preprocessing

The preprocessing in the case of the 3D segmentation is pretty much the same as for the 2D segmentation but expands the methods by an additional dimension. The five preprocessing approaches in 3D:

- Random patch extraction: I have only used cubic patches of sizes $32 \times 32 \times 32$ and $64 \times 64 \times 64$ because the network could not cope with bigger patches.

  I have also used the two mechanisms described in 2.2.1 for the random patch extraction. Both mechanisms extract cubes randomly around pixels, where there is a vessel present and extract the accompanying masks. Therefore all extracted

cubes contain vessels, but not all slices from the cubes, because some vessels end or start right in between the cube. The general idea of the 3D patch extraction is depicted in figure 2.7.

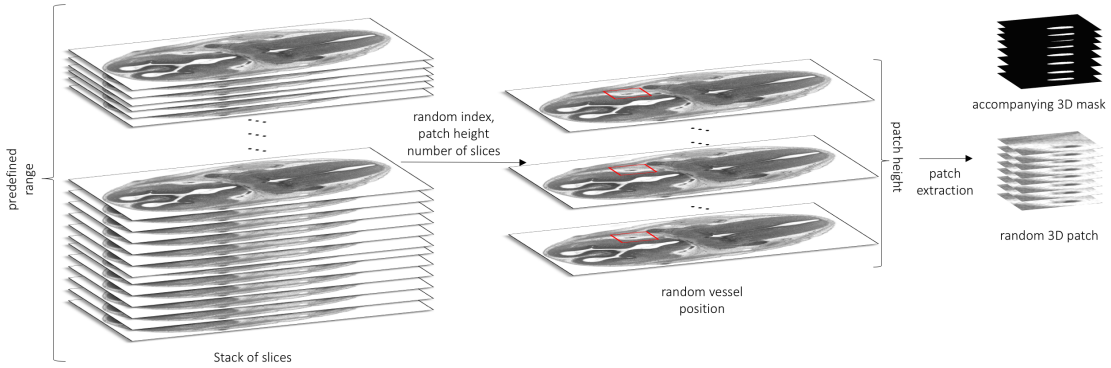


Figure 2.7: Random patch extraction in 3-dimensions.

In 3D I have experimented more with generators because the dynamic creation of 3D patches and accompanying masks let me train on more examples. The best results, however, were achieved with a static stack of cubes as shown in chapter 3.

- Zero-mean unit variance normalization: I have also used patch-wise normalization in three dimensions, which results from subtracting the mean of the whole 3D patch and subsequently dividing the patch by its respective standard deviation.
- Downsampling: The images were also downsampled by a factor of 2 by only taking every second voxel value in each dimension and discarding the rest to get more context information in the cubes.
- Denoising: In 3D the network seems to have more troubles with noisy data, although the additional dimension gives more contextual information. I have also used the *median filter* in three dimensions, which takes the median of a specified three-dimensional window, for example, a $3 \times 3 \times 3$ window.
- Big vessels: Also in the 3D segmentation task, I have experimented with discarding the patches with the least 25%, 20%, 15% or 10% representation of vessels.

2.3.2 3D U-Nets

I have used the 3D version of the U-Net shown in section 2.2.2 as the primary deep learning algorithm from Özgün Çiçek et al. [31] from the year 2016.

The original architecture is shown in figure 2.8 with an input of $132 \times 132 \times 116$ pixel images and 3 channels. The network uses fewer convolution filters due to the increasing network size. With a maximum of 256 feature maps at the "bottom" of the network, the architecture contains 19069955 parameters. It uses the same activation and loss functions,

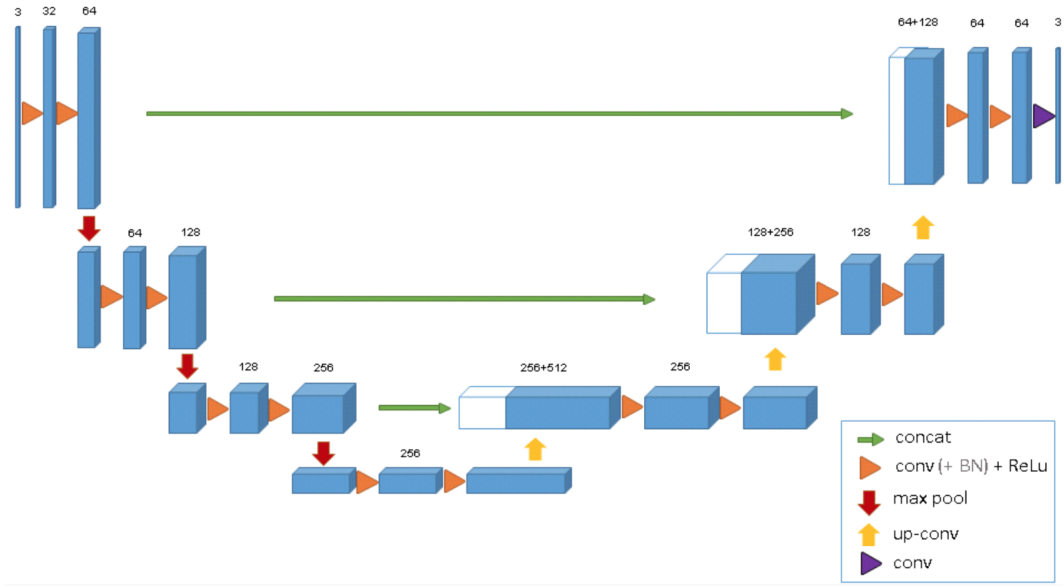


Figure 2.8: U-Net 3D [31].

as the two-dimensional version, but introduces a batch normalization before the rectified linear units.

The work from Özgün Çiçek et al. [31] was concerned with learning dense volumetric segmentation from sparse annotation, which differs from this thesis' objective, where a general segmentation of blood vessels should be achieved.

2.3.3 Project Set-Up

The 3D segmentation algorithm is very similar to the 2D segmentation implementation. It starts by saving a range of the original slices and converts the accompanying labels to ground truth masks. For the patch extraction, I used the self-written generator functions, which have different parameters for preprocessing, like the size of patches, denoising, the portion of vessels tissue in a cubic patch, the downsampling rate, etc. all in three dimensions.

The stack of patches or the generators is then sanity checked. I have plotted slices of the extracted cubes, checked unit variance and zero mean, created scatter plots to visualize the pixel values, etc.

Afterwards, the program builds the 3D U-Net, which is an extension of the 2D model by applying 3D convolutions with $3 \times 3 \times 3$ filters, up-sampling in 3D, using an extended version of the weighted pixel-wise cross entropy loss function, etc. The activation functions and the metrics for accuracy remain the same. The architecture with 512 feature maps at the "bottom" of the network has a total of 23532322 trainable parameters. If the

network would convolute up to 1024 feature maps, the model would contain over 90 million trainable parameters, which is too much for the external GPU for training with bigger batch-sizes.

The training itself and the testing are all extended to the 3D versions of Keras. To predict a whole image of the embryo data, the HREM slices have to be stacked together to form a stack of as many slices as the height of the patches. Then the cubes are extracted from the stack of slices in a similar manner as it is depicted in figure 2.6.

CHAPTER 3

Results

All together I have conducted over 40 experiments in 2D and over 30 experiments in 3D. In this chapter, I will describe the top 5 experiments of the 2D and the top 3 experiments of the 3D segmentation respectively. I will explain the parameters I have used and what generally worked well and what turned out to work badly. For the best results, the predictions are shown in 2D and 3D. I will only reference the first mice embryo data set with 2857 slices because the results of the other data set were very similar.

Section 3.3 discusses issues I have encountered, gives some recommendations for further experiments, which I could not complete because this would go beyond the scope of a Bachelor thesis, and gives practical information about computing time.

3.1 Results for 2D Segmentation

What proved to be important for the U-Net in the case of 2D segmentation were the *patch-size*, the *batch-size* and the *weighted pixel-wise cross entropy*. The patch-size proved to be best at 64×64 pixels with a large batch-size of between 64 and 100. Smaller batch-sizes caused the network to stop learning at an early stage because the network learned individual features at certain locations too soon. With bigger patch-sizes, the network had problems with noisy patches and small tubular structures.

To reduce the probability of predicting noise as small vessels, the 75% largest portions of vessels tissue in patches have been selected for training. The best number of convolutional layers proved to be the same as Ronneberger et al. used in their paper [28] with 1024 feature maps at the "bottom" of the network, which resulted in 31377858 trainable parameters.

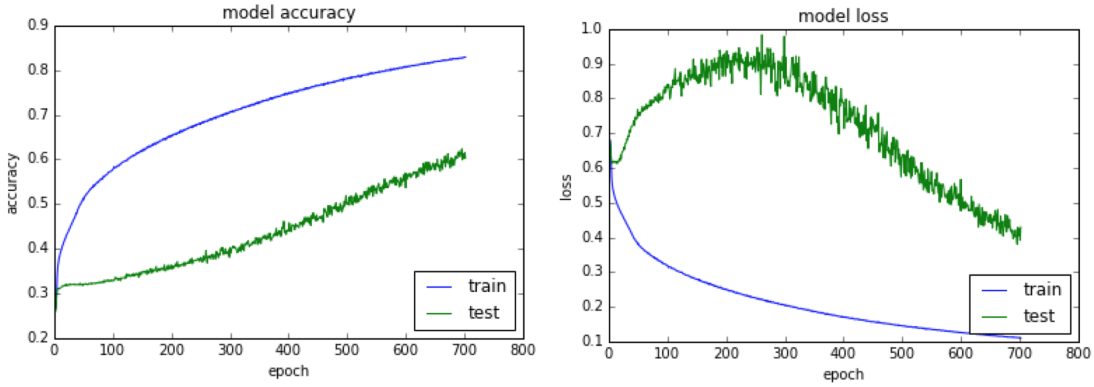
Weighted pixel-wise cross entropy loss function and pixel-wise softmax as the core activation function were superior to the dice loss function from equation 2.1, which uses the dice score for calculating the error function and the sigmoid activation. Also, the

optimizer Adam [29] yielded better results than the built-in stochastic gradient descent algorithm from Keras. Table 3.1 shows the top 5 experiments for the 2D segmentation task.

Table 3.1: Results in 2D

Experiment (No.)	**Batch size (No.)**	**Discard (%)**	**Loss (Train/Val/Test)**	**Dice1 (Train/Val/Test)**
1	100	0	0.11/0.42/0.56	0.84/0.65/0.61
2	100	25	0.15/0.61/0.72	0.87/0.63/0.59
3	100	25	0.12/0.25/0.35	0.74/0.52/0.51
4	64	0	0.21/0.35/0.92	0.65/0.55/0.36
5	64	0	0.25/0.38/1.21	0.61/0.46/0.32

- Experiment 1: The best result of my 2D experiments yielded an accuracy, measured for $dice_1$, on the training set of 84%, on the validation set of 65% and on never before seen test data of 61%, which is shown in figure 3.1. This is a clear case of overfitting, although the dropout rate was set to 0.2 after each convolution layer.



(a) Accuracy history of the model.

(b) Loss history of the model

Figure 3.1: History of experiment 1.

For experiment 1 the network was built from 8000 patches of the size 64×64. The patches were extracted randomly around vessels and started from index 800. The stack of patches was split with a ratio of 1:5 = validation:training set. The architecture with 31377858 and 1024 convolution filters at the "bottom" of the U-Net was used with ReLU activation functions after each convolution, a pixel-wise softmax after the final layer and a weighted pixel-wise cross entropy for the loss function. The learning rate for Adam [29] was set to 10^{-6} and the batch-size was set to 100. With 50 seconds training time per epoch, the model has trained 10.5 hours until the loss turned to a "nan"-error. A discussion on why this error has occured can be found in section 3.3.

Figure 3.2 shows random image patches, the accompanying ground truth masks and the predictions of the previously unseen test data. The predictions were thresholded at 0.5 and are displayed as a binary masks. Ground truth masks and predictions have been inverted, where white represents vessel tissue and black non-vessel tissue.

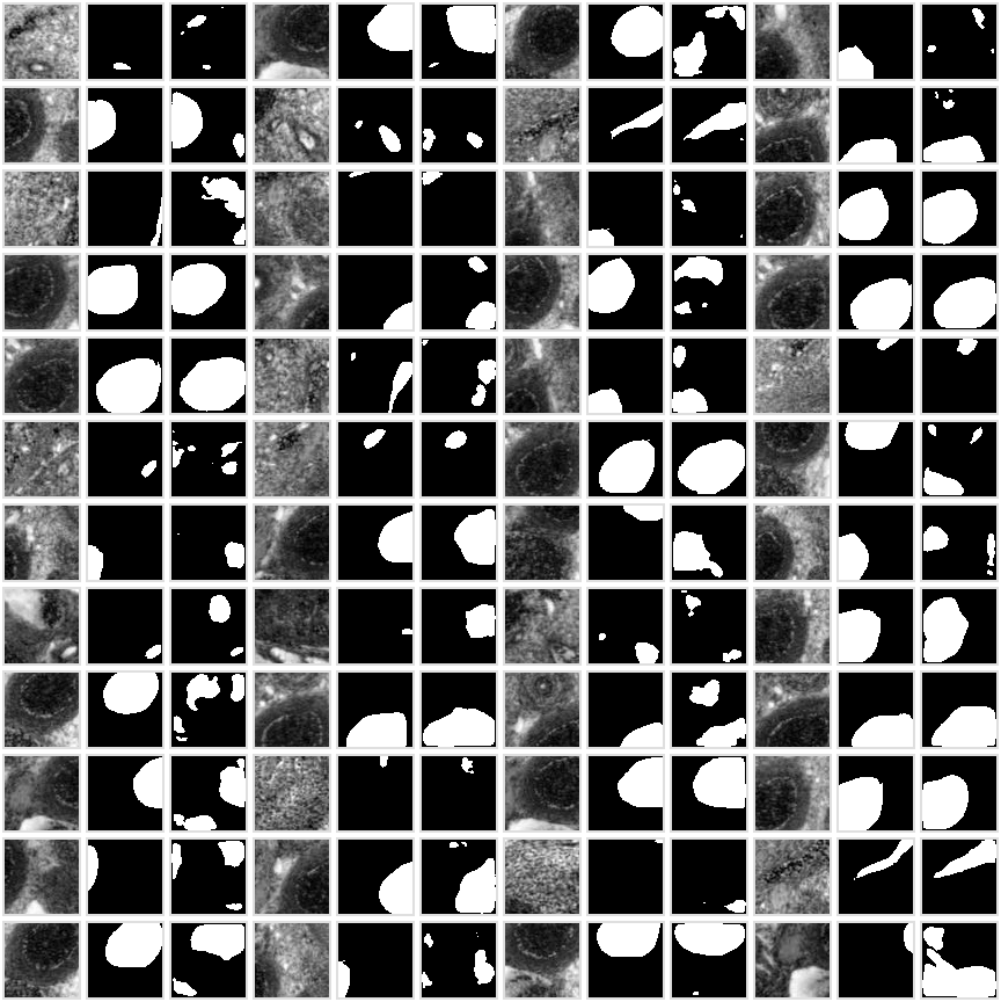


Figure 3.2: Image patches, accompanying ground truth masks and the predictions in 2D from experiment 1.

Figure 3.2 shows good predictions for very large vessels, however, uncertainties for noisy patches and has difficulties with similar examples to figure 3.3, which shows how different vessel structures can be. Figure 3.3 displays the predictions as a colourmap, where dark red colours display values near 1, while dark blue colours display values near 0. The network has problems with lighter vessels because they are underrepresented in the embryo study. Moreover, when looking at figure 3.3a, there are three very similar structures of which only one is segmented as a vessel.

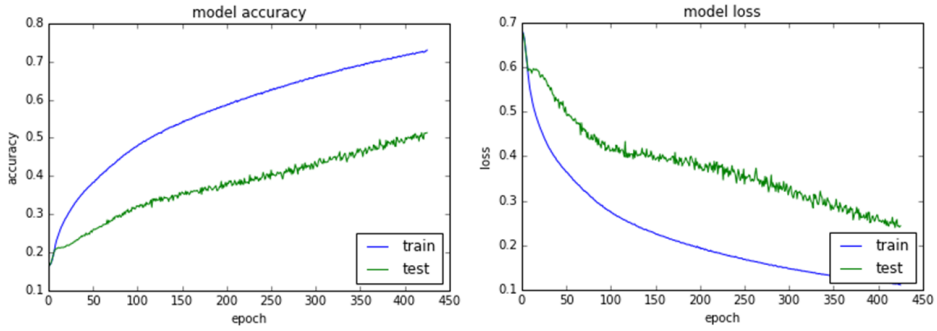
Thoughts on how to improve performance tailored to these issues are presented in section 3.3.

(a) Light vessels. (b) Dark vessels

Figure 3.3: Patch(left), ground truth mask(middle), prediction(right).

- Experiment 2: The second best result of my experiments yielded an accuracy, measured for $dice_1$, on the training set of 87%, on the validation set of 63% and on never before seen data of 59%. It uses only the biggest 75% of the vessels, but apart from that the same settings from experiment 1. With 50 seconds training time per epoch, the model has trained 10.6 hours until the loss turned to a "nan"-error.

- Experiment 3: In this experiment, I have downsampled the original stack of images by a factor of 2 and afterwards started to extract random 64×64 patches. The other parameters were set to the same values as for experiment 1 and 2. The network showed a similar accuracy curve as depicted in figure 3.4a, but a more stable loss history, which is shown in figure 3.4b. The experiment yielded an accuracy, measured for $dice_1$, on the training set of 74%, on the validation set of 52% and on never before seen test data of 51%. The model trained only for 6 hours because after 430 epochs the loss turned "nan".



(a) Accuracy history of the model. (b) Loss history of the model

Figure 3.4: History of experiment 3.

- Experiment 4: One of my first experiments yielded an accuracy, measured for $dice_1$, on the training set of 65%, on the validation set of 55% and on never before seen data of 36%. In this experiment, I have only extracted 2048 64×64 patches for training, starting from index 600 and used a batch-size of 64. The model used a learning rate for Adam of 10^{-7} and trained within 4 seconds per epoch in a stable way until epoch 1500, where the model stagnated. The model recognizes simple

structures but has difficulties with smaller and more complex vessels, which results from the relatively small training set. Some predictions are shown in figure 3.5.

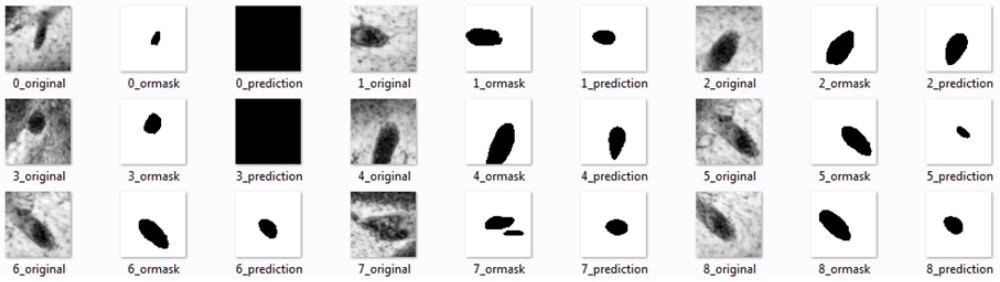


Figure 3.5: image patches, accompanying ground truth masks and the predictions in 2D from experiment 4.

- Experiment 5: The same training parameters from experiment 4 were also used in this experiment, with a higher learning rate of 10^{-6} for Adam. It yielded an accuracy, measured for $dice_1$, on the training set of 61%, on the validation set of 46% and on never before seen data of 32%.

3.2 Results for 3D Segmentation

Changing the *batch-size*, the *downsampling* when preprocessing, the *denoising* with the median filter and the *out-filtering of small vessels* showed significant changes in the network's learning history.

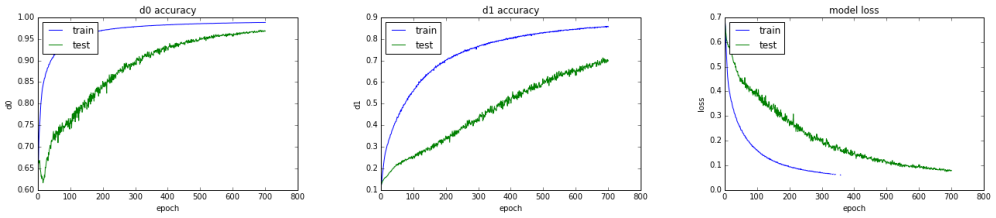
The biggest batch-size the external GPUs could process was 10, due to the additional dimension of the patches. The downsampling by a factor of 2 turned out to make good improvements because the contextual information increased and the noise was reduced. To further reduce the probability of predicting noise as small vessels, the 75% biggest vessels have been selected for training and the median filter with a $3 \times 3 \times 3$ window was used.

Similar to the 2D case, the weighted pixel-wise cross entropy loss function and pixel-wise softmax as the core activation function were superior to the *dice* loss function and the sigmoid activation. Also, the optimizer Adam [29] yielded better results in 3D, than the built-in stochastic gradient descent algorithm from Keras. Table 3.2 shows the top 3 experiments for the 3D segmentation task.

Table 3.2: Results in 3D

Experiment (No.)	**Learning rate (Adam)**	**Dropout (%)**	**Loss (Train/Val/Test)**	**Dice1 (Train/Val/Test)**
1	10^{-6}	30	0.05/0.08/0.06	0.87/0.71/0.66
2	10^{-5}	20	0.05/0.10/0.20	0.77/0.55/0.54
3	10^{-5}	20	0.06/0.11/0.21	0.83/0.61/0.53

- Experiment 1: The best results of my experiments in 3D yielded an accuracy, measured for $dice_1$, on the training set of 87%, on the validation set of 71% and on never before seen data of 66%. The $dice_0$ accuracy was 99% on the training set and 96% on the validation set. The history can be seen in figure 3.6.



(a) $Dice_0$ accuracy. (b) $Dice_1$ accuracy. (c) Loss.

Figure 3.6: History of experiment 1 in 3D.

The original image stack has been downsampled by a factor of 2 and median-filtered. Of this stack, I have extracted 1200 patches of the size $64 \times 64 \times 64$, of which I have discarded 300 because I only trained on the biggest 75% of the vessel data. The

patches were extracted randomly around vessels and started from slice index 800. The stack of patches was split with a ratio of 1:5 = validation:training set. The architecture used ReLU activation functions after each convolution, a pixel-wise softmax after the final layer and a weighted pixel-wise cross entropy for the loss function. The learning rate for Adam was set to 10^{-6}. The Dropout for this experiment was set to 0.3, which yielded better generalization for the problem. A single epoch took 183 seconds. As the model's accuracy turned to "nan" after approximately 700 epochs, the model has trained for 35.5 hours.

The predictions shown are almost identical and in many cases even more appealing than the ground truth masks and yield very good results on training, validation and test sets. They show different examples of random patch slices, because they are three-dimensional patches, accompanying masks and predictions. Figure 3.7 to figure 3.11 show different examples from the training set and figure 3.12 to figure 3.16 show examples from never before seen test data. The predictions have been thresholded with a threshold of 0.9.

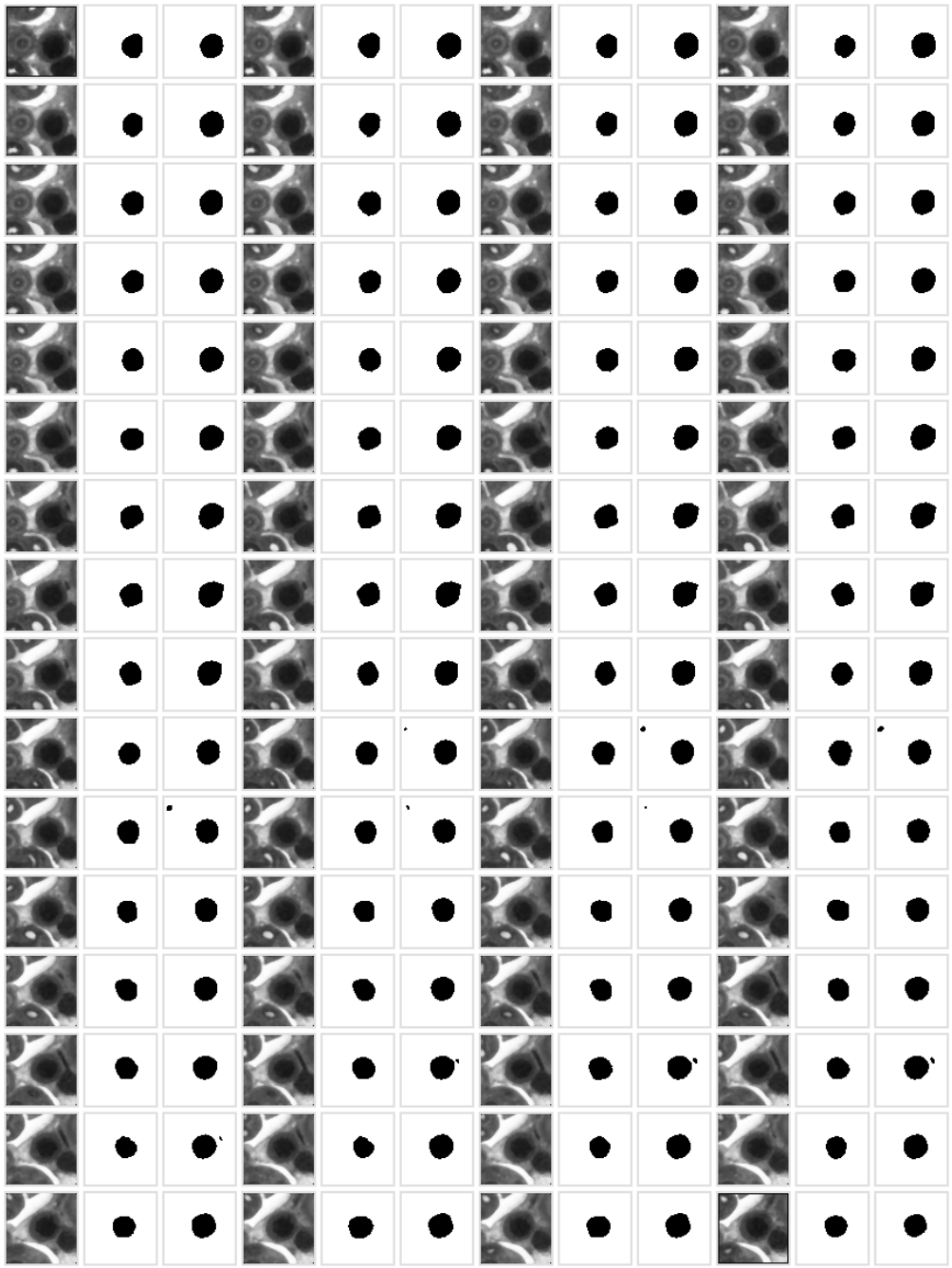


Figure 3.7: Example of dark gray circular vessels in training patch slices, accompanying ground truth masks and the predictions in 3D from experiment 1.

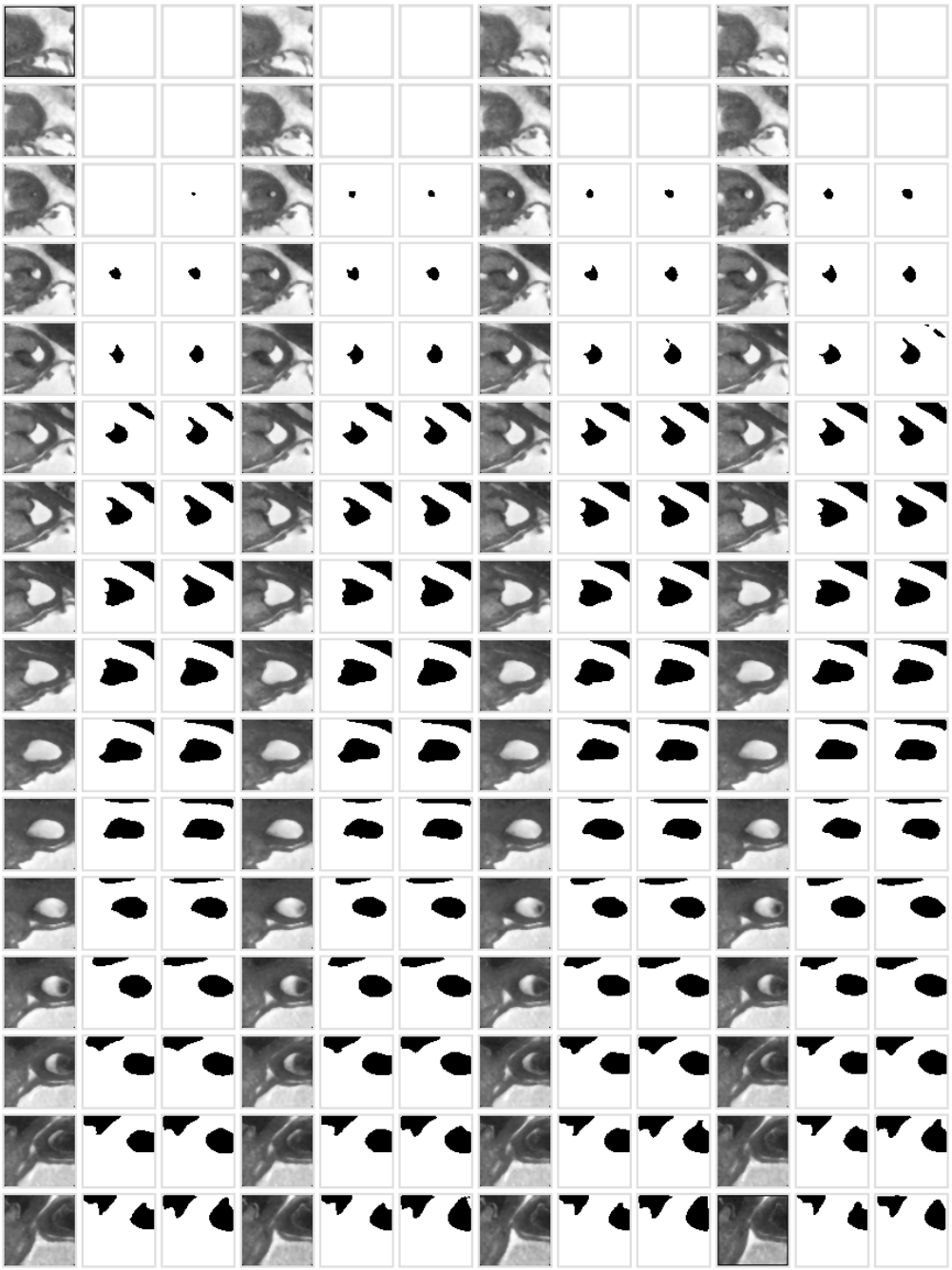


Figure 3.8: Example of ligth and dark gray vessels in training patch slices, accompanying ground truth masks and the predictions in 3D from experiment 1.

Figure 3.9: Example of gray circular vessels in training patch slices, accompanying ground truth masks and the predictions in 3D from experiment 1.

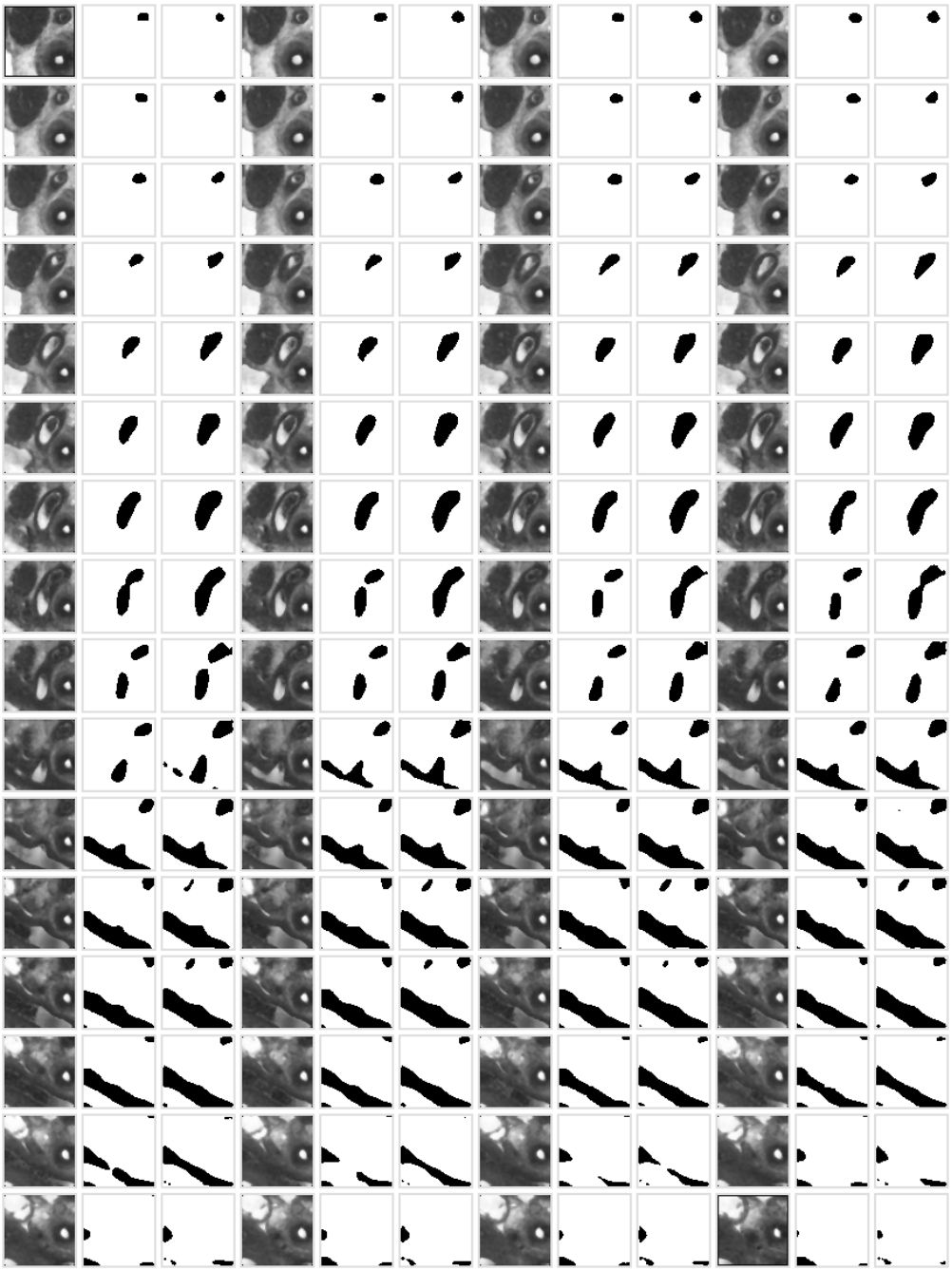


Figure 3.10: Example of branching vessels in training patch slices, accompanying ground truth masks and the predictions in 3D from experiment 1.

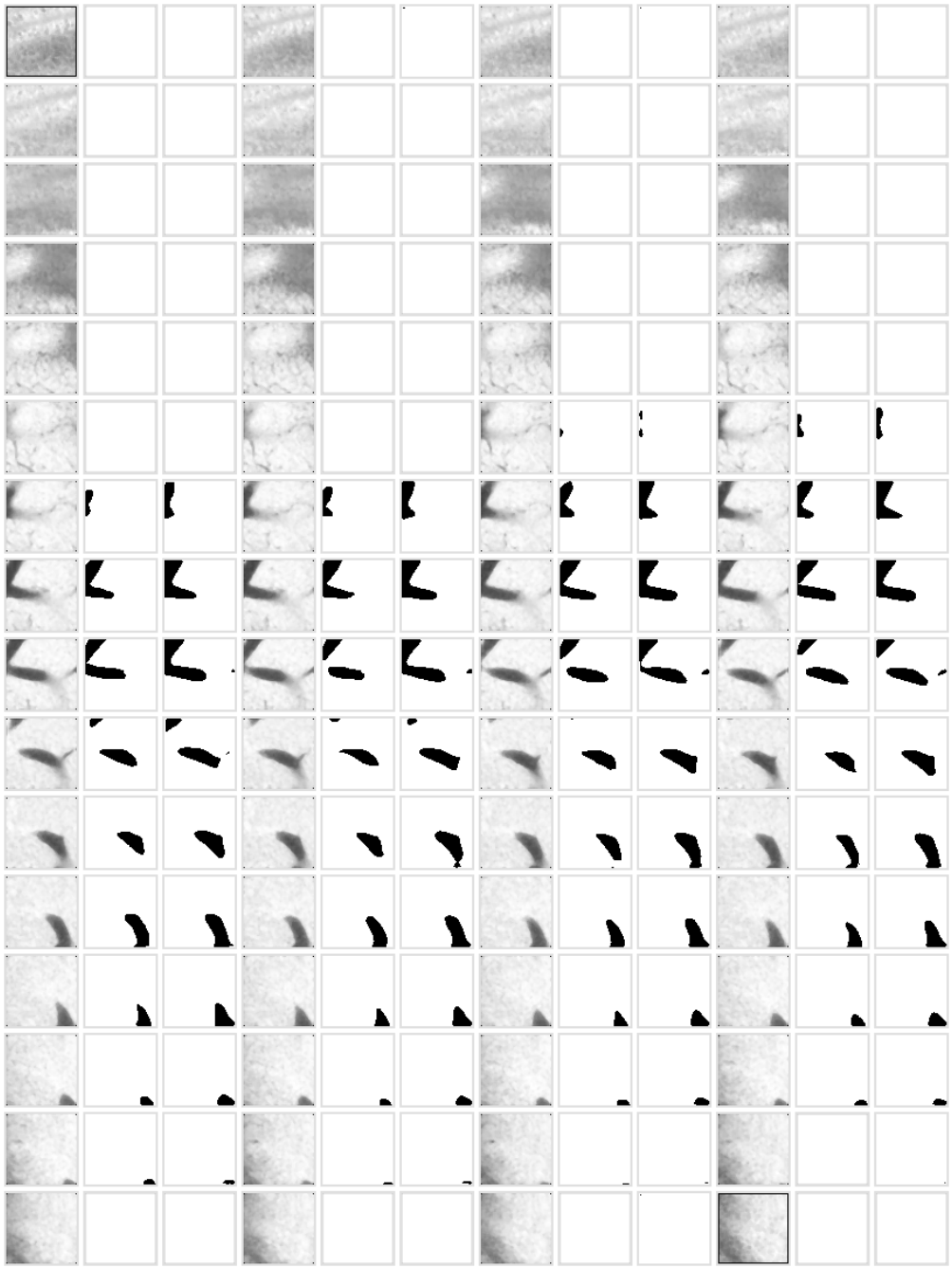


Figure 3.11: Example of vessels in the center of the training patch slices, accompanying ground truth masks and the predictions in 3D from experiment 1.

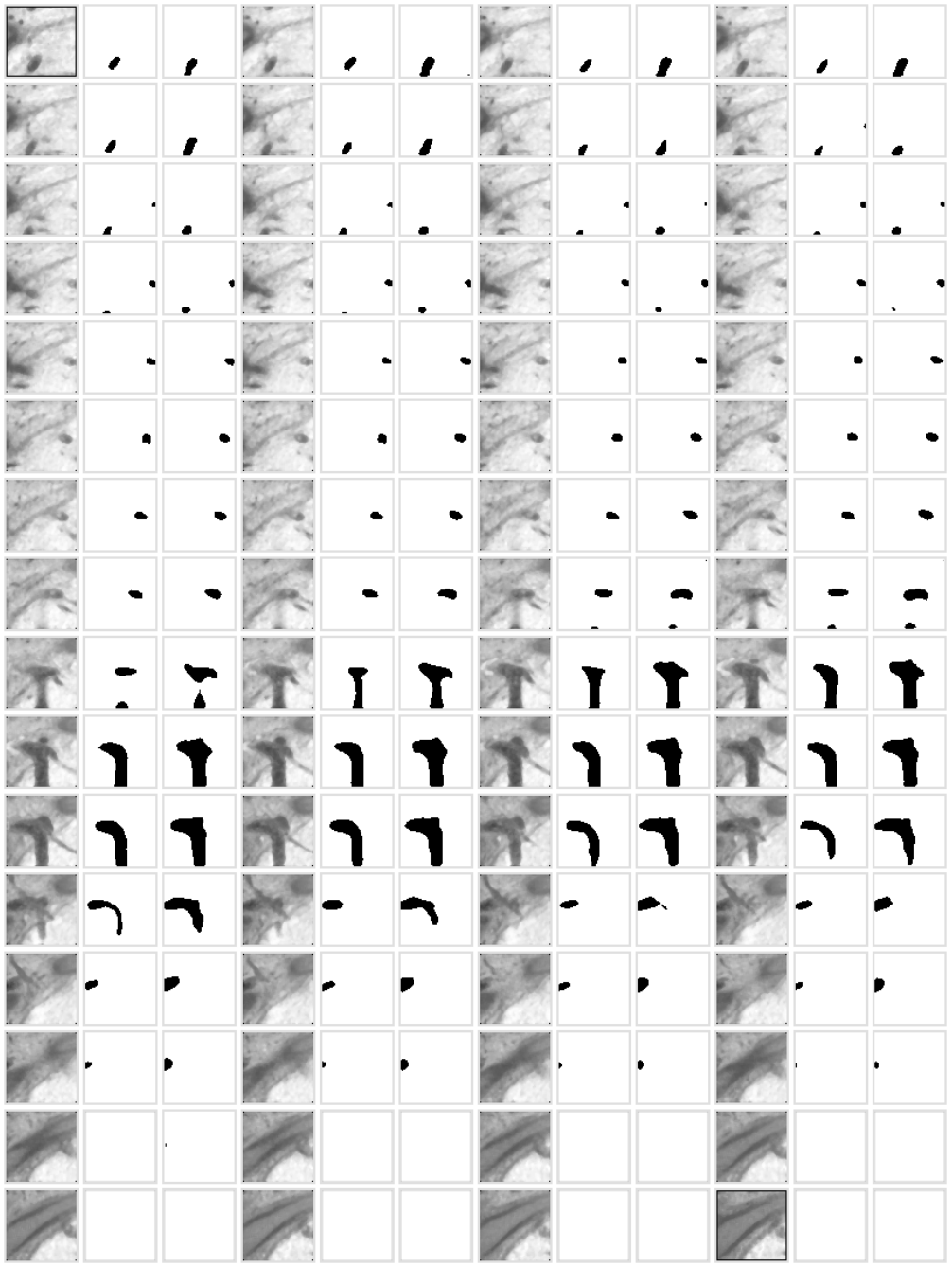


Figure 3.12: Example of gray branching vessels in test patch slices, accompanying ground truth masks and the predictions in 3D from experiment 1.

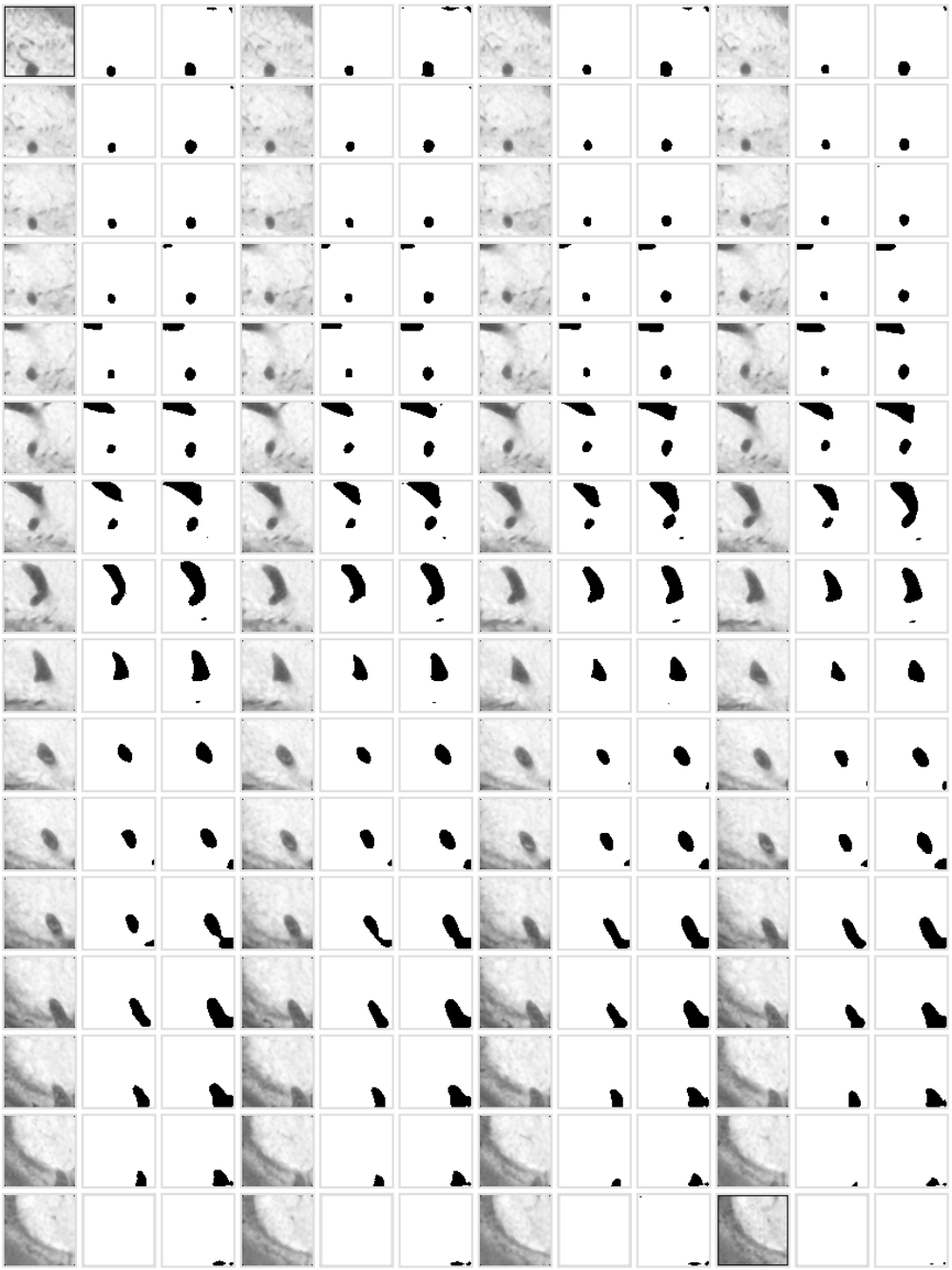


Figure 3.13: Example of gray branching vessels in test patch slices, accompanying ground truth masks and the predictions in 3D from experiment 1.

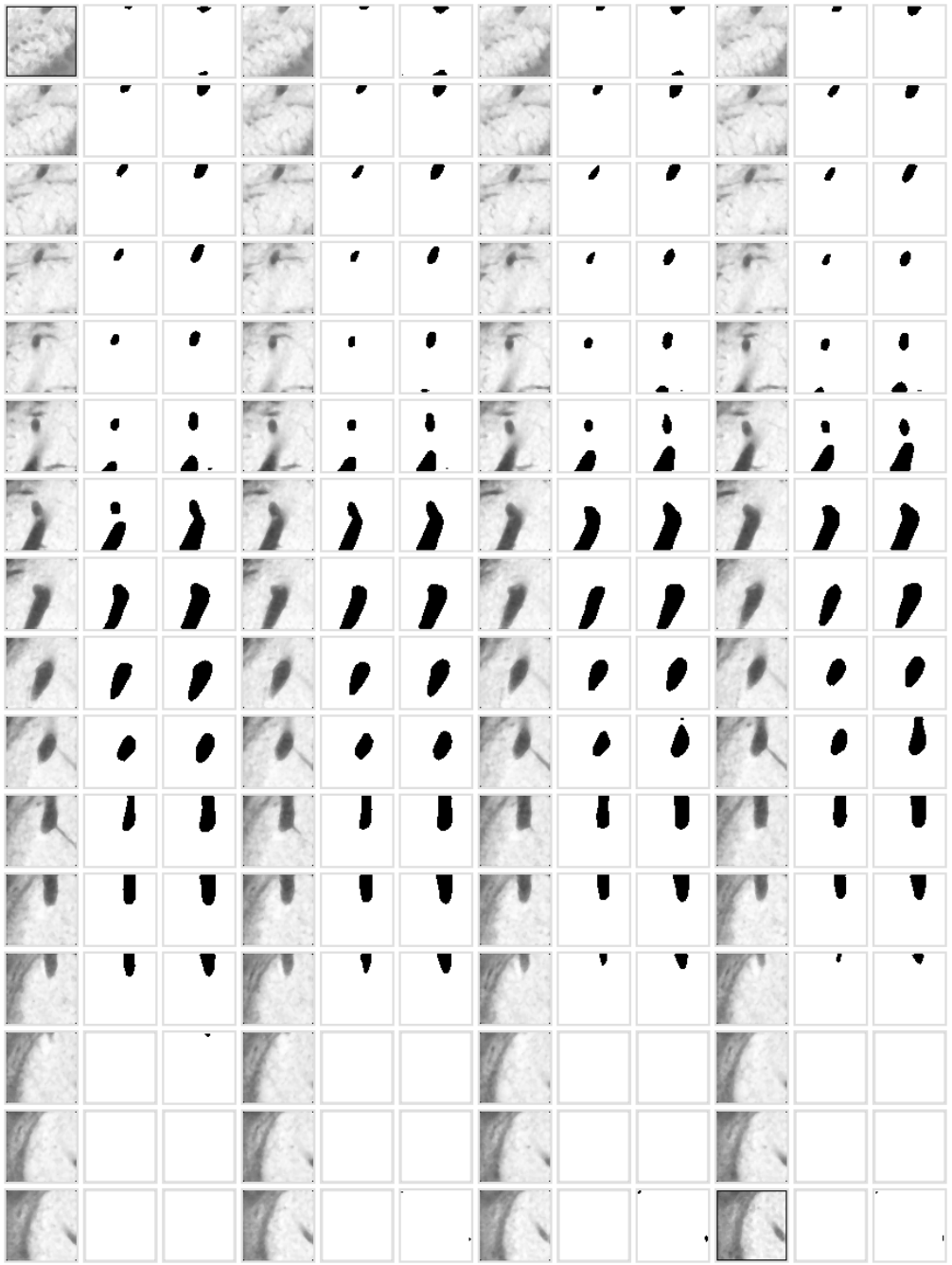


Figure 3.14: Example of gray combining vessels in test patch slices, accompanying ground truth masks and the predictions in 3D from experiment 1.

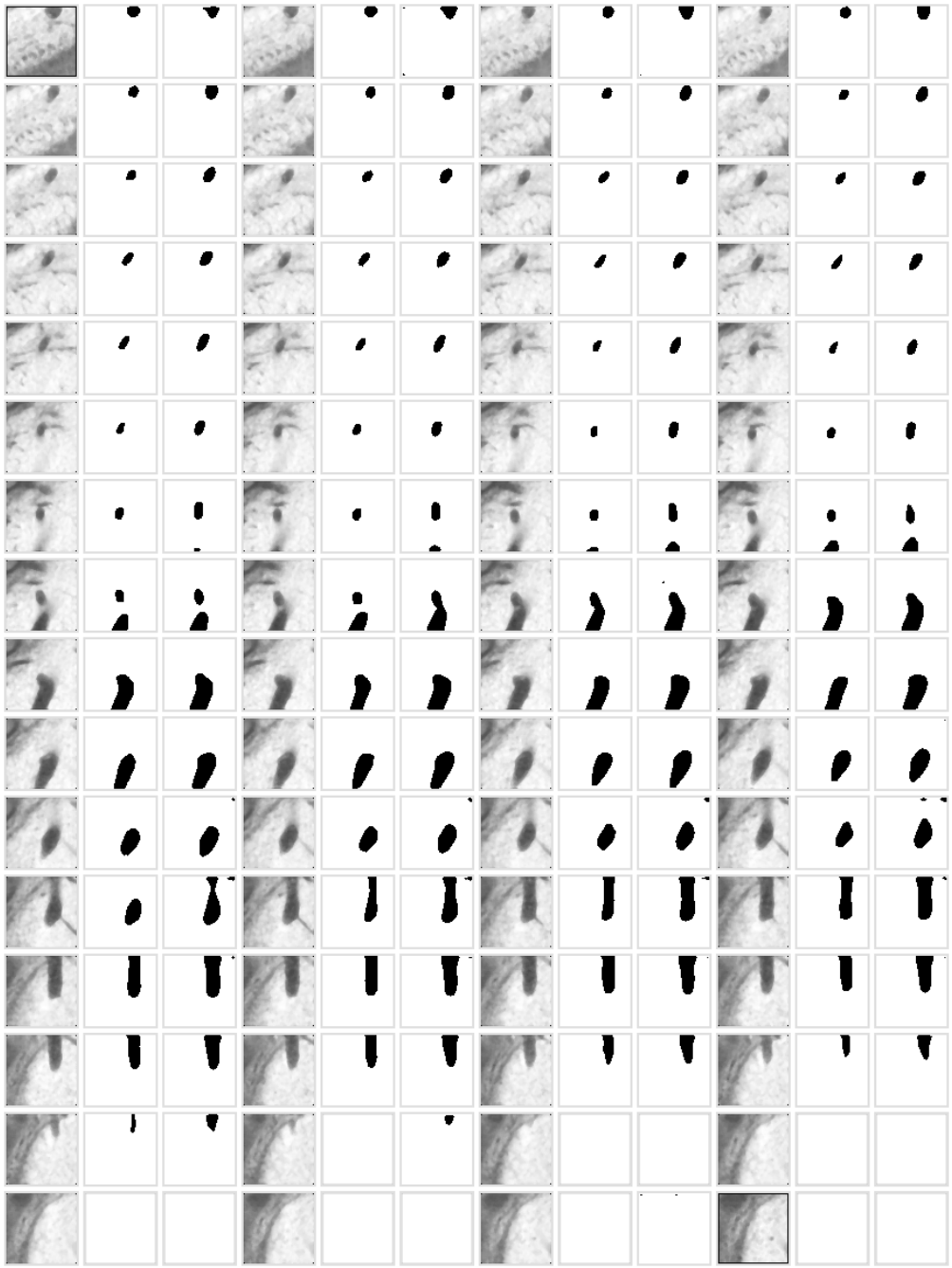


Figure 3.15: Example of gray combining vessels in test patch slices, accompanying ground truth masks and the predictions in 3D from experiment 1.

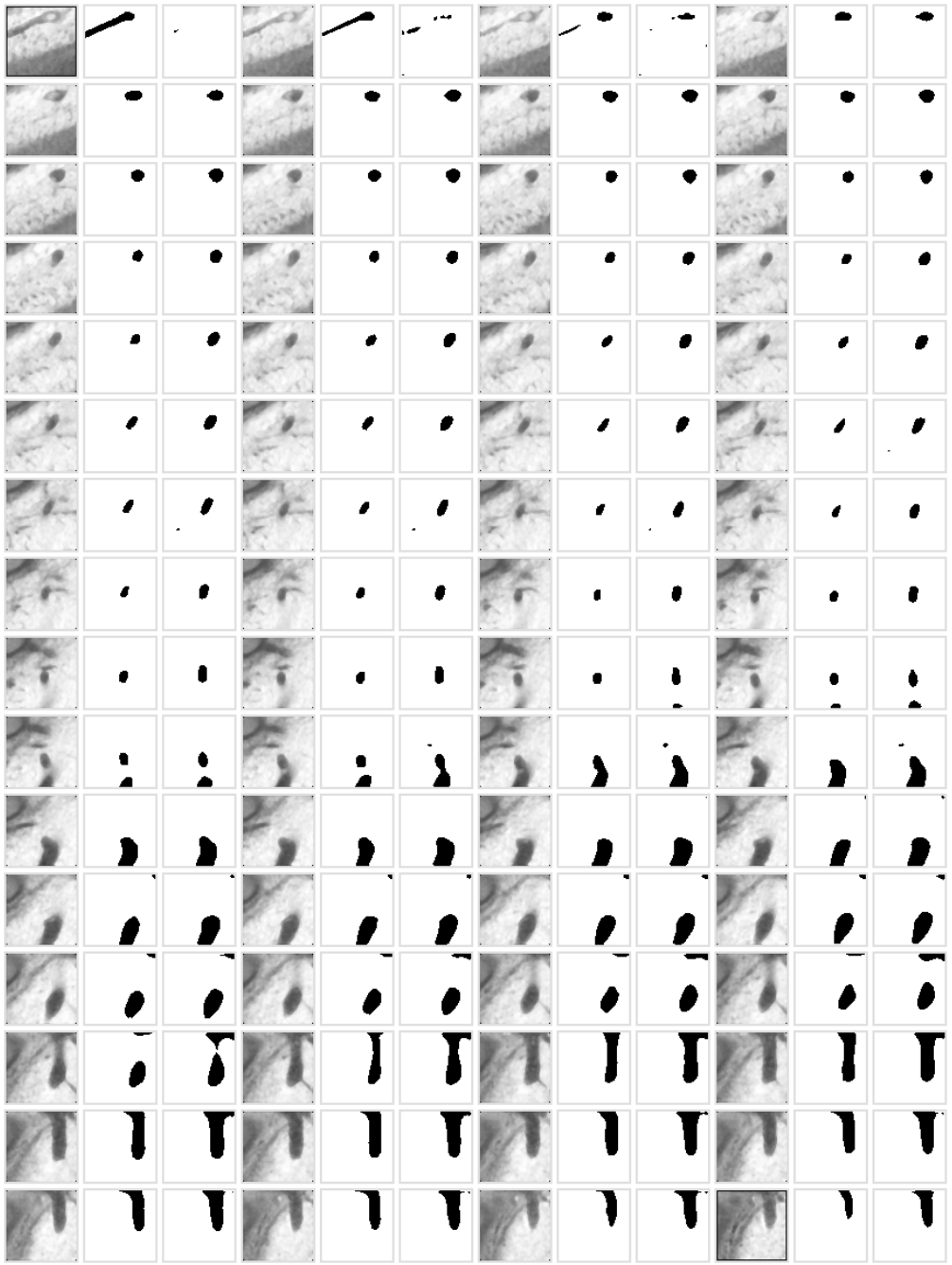
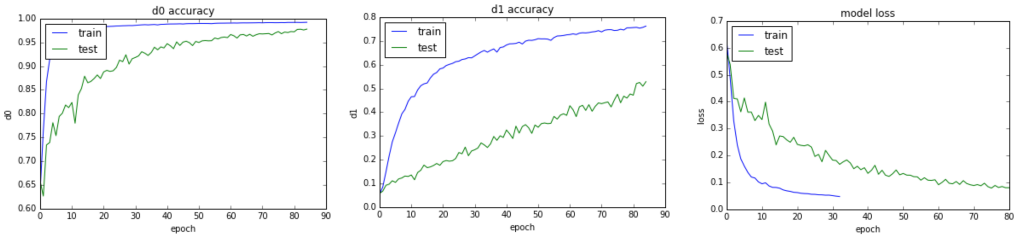


Figure 3.16: Example of gray combining vessels in test patch slices, accompanying ground truth masks and the predictions in 3D from experiment 1.

- Experiment 2: The second best result of my experiments in 3D yielded an accuracy, measured for $dice_1$, on the training set of 77%, on the validation set of 55% and on never before seen data of 54%. The $dice_0$ accuracy was 99% on the training set and 97% on the validation set. The history can be seen in figure 3.17.

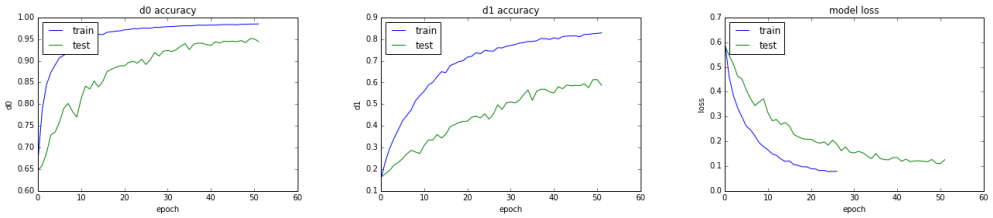


(a) $Dice_0$ accuracy. (b) $Dice_1$ accuracy. (c) Loss.

Figure 3.17: History of experiment 2 in 3D.

The model used images, which have not been median-filtered, applied a dropout of 0.2 after each convolution and a learning rate for Adam of 10^{-5}. The other settings were the same as for experiment 1.

- Experiment 3: In this experiment, I have again used median filtered images and extracted 1200 patches, of which 25% with the least vessel representation were discarded. The images were also downsampled by a factor of 2 and yielded an accuracy, measured for $dice_1$, on the training set of 83%, on the validation set of 61% and on never before seen data of 53%, which can be seen in figure 3.18. The other settings are the same that were used in experiment 2.



(a) $Dice_0$ accuracy. (b) $Dice_1$ accuracy. (c) Loss.

Figure 3.18: History of experiment 3 in 3D.

3.3 Discussion and Future Work

The denoising of the data with the median filter showed good results for the three-dimensional segmentation task. I have conducted experiments with denoising at the very end of my time at the VRVis, which is the reason, why I would continue with different denoising methods. An overview of image denoising algorithms is given in the work of Buades et al. [32]. Some often used methods are k-nearest neighbours filters [33], non-local means filter [34], Gaussian filter [34], wavelet coefficient denoising [35] and many more. There are also approaches, which use deep neural networks for image denoising, like the paper from Xie et al. [36] or the paper on "Medical image denoising using convolutional denoising autoencoders" from Lovedeep Gondora [37].

Additional and promising approaches for preprocessing of data are the *principal component analysis (PCA)* [38] and whitening of data [39]. PCA reduces the dimensions of the data by using an orthogonal transformation that only keeps those dimensions with maximal variance and whitening is a normalization technique.

Another preprocessing or data augmentation approach I would recommend is the inversion of patches. As seen in figure 3.3 there are examples, which are nearly the opposite of one another. Inverting the inputs could result in better learning of these structures.

As the weighted pixel-wise cross entropy loss function showed improved accuracy, I would increase the weighting of the vessels.

To prevent overfitting, the dropout rate can be increased or of course, more data could be trained on. I have mostly experimented with dropout rates ranging from 0.2 to 0.3. As the generators provide practically endless patches in a relatively large range of images, data augmentation techniques, despite the inversion of the patches, are not necessary.

The current classification problem is concerned with two classes, one class for vessel and one class for non-vessel tissue. The network has problems at the transition from the mice embryo's body to the background because the training patches in my experiment were exclusively taken from structures in between the body interior. To improve the prediction accuracy at this transition, the problem could be formulated as a three class problem, where everything outside the embryo's body is classified as background and the rest as vessel or non-vessel tissue.

Another approach for solving the prediction problems at the transition from the mice embryo's body to the background would be to use a vessel tracker first, like the one presented by Felkel et al. [40]. Around the tracked vessel locations, patches could be extracted and the trained model would have to predict the exact structure of the vessels inside these patches.

With Adam's learning rate between 10^{-5} and 10^{-6} the network trains very fast within a few hours up to the results shown in table 3.2. A slower learning rate might yield better results because the gradient descent algorithm takes smaller steps.

The results from section 3.1 and section 3.2 show "nan" for loss and accuracy after some epochs. There are several discussions on "nan" like Russel Stewart's notes on this topic [41]. One of the suggestions from these notes is concerned with high learning rates, which occurs with my networks if Adam's learning rate is set to a value equal or above 10^{-5}. Another suggestion on why these "nan" values occur, is concerned with self-written loss and activation functions. I have used self-written implementations of the pixel-wise softmax activation function and the weighted pixel-wise cross entropy loss function. I have checked the implementations for numerical issues, like the division of 0 or a logarithm of 0, etc. The softmax activation function could return a "nan" if the network would predict only zeros, which does not occur in practice. The cross entropy loss could return a "nan" if the prediction of any pixel/voxel is 0 because it calculates the logarithm of these values. This is more plausible because the loss turns to "nan" only after some epochs. Therefore, I have implemented a clipping operation, which thresholds the predictions, such that the predictions can never be 0. This, however, had no positive effects on the network's learning history. I have also sanity-checked the weights of a trained model, which shows no anomalies. I am not sure, why these "nan" values occur after some hundred epochs, but I would suggest to reimplement the loss and activation functions and use slower learning rates, like 10^{-6} or 10^{-7}.

Another issue of some results is the drop between validation and test scores. If the test patches are extracted from the range, as the training patches, this drop is relatively small and shows good generalization results. If the test patches are, however, extracted from another range, the mice embryo body appears to have slightly different structures. I would therefore recommend, that the model should be trained on the mice's whole body interior and not only on certain ranges. The $dice_1$ scores seem to be numerically worse than visually, as depicted in some example predictions of the last two sections. The definition of the dice coefficient is responsible for this. If the network predicts some additional tissue around the ground truth mask, which seems visually plausible, the dice score gets reduced, because only overlapping values count.

The training time ranges between a few hours with a high learning rate and up to 48 hours with lower learning rates and 1000 epochs. After the model has been trained, the prediction of the downsampled data of size $1427 \times 1140 \times 877$ with my self-written functions would take about 11 minutes or for the original data set 88 minutes. The median-filtering of the whole set of size $1427 \times 1140 \times 877$ with SciPy's *medfilt* function takes approximately 45 minutes because the window of the filter has to be slid across every voxel. A fast implementation of the median filter operation is given by Perreault and Hébert [42].

All together I am convinced that the results on this data can be improved by a lot of fine-tuning, which would go beyond the scope of a Bachelor thesis. For this thesis I have described the fundamentals for understanding the algorithms of chapter 2, I have implemented two popular and modern deep learning algorithms in Python and have trained a machine so that it can learn tubular structures from HREM data in two and in three dimensions.

List of Figures

List of Tables

Bibliography

[1] W. J. Weninger, S. H. Geyer, T. J. Mohun, D. Rasskin-Gutman, T. Matsui, I. Ribeiro, L. d. F. Costa, J. C. Izpisua-Belmonte, and G. B. Müller, "High-resolution episcopic microscopy: A rapid technique for high detailed 3d analysis of gene activity in the context of tissue architecture and morphology," *Anatomy and embryology*, vol. 211, no. 3, pp. 213–221, 2006.

[2] A. Munoz, "Machine learning and optimization," *https://www.cims.nyu.edu/~munoz/files/ml_optimization.pdf*, 2014. Accessed: 2017-08-07.

[3] T. M. Mitchell, "Machine learning. 1997," *Burr Ridge, IL: McGraw Hill*, vol. 45, no. 37, pp. 870–877, 1997.

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[5] *MNIST data set*. http://yann.lecun.com/exdb/mnist/. Accessed: 2017-08-07.

[6] D. C. Ciresan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," *CoRR*, vol. abs/1202.2745, 2012.

[7] *Scikit over- and underfitting example*. http://scikit-learn.org/stable/auto_examples/model_selection/plot_underfitting_overfitting.html. Accessed: 2017-08-07.

[8] H. Fischer and H. Kaul, "Lineare algebra," in *Mathematik für Physiker*, pp. 284–387, Springer, 2011.

[9] E. L. Lehmann and G. Casella, *Theory of point estimation*. Springer Science & Business Media, 2006.

[10] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting.," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[11] T. Hastie, R. Tibshirani, and J. Friedman, "Overview of supervised learning," in *The elements of statistical learning*, pp. 9–41, Springer, 2009.

[12] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, vol. 1. MIT press Cambridge, 1998.

[13] D. Graupe, *Principles of artificial neural networks*, vol. 7. World Scientific, 2013.

[14] *Stanford University Computer Science Course CS231n neural networks.* `http://cs231n.github.io/neural-networks-1/`. Accessed: 2017-08-07.

[15] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015.

[16] K. Janocha and W. M. Czarnecki, "On loss functions for deep neural networks in classification," *CoRR*, vol. abs/1702.05659, 2017.

[17] *Stanford University Computer Science Course CS231n.* `http://cs231n.github.io`. Accessed: 2017-08-07.

[18] R. A. Dunne and N. A. Campbell, "On the pairing of the softmax activation and cross-entropy penalty functions and the derivation of the softmax activation function," in *Proc. 8th Aust. Conf. on the Neural Networks, Melbourne, 181*, vol. 185, 1997.

[19] P.-T. De Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein, "A tutorial on the cross-entropy method," *Annals of operations research*, vol. 134, no. 1, pp. 19–67, 2005.

[20] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[21] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*, pp. 177–186, Springer, 2010.

[22] *Stanford University Computer Science Course CS231n convolutional neural networks.* `http://cs231n.github.io/convolutional-networks/`. Accessed: 2017-08-07.

[23] N. Senthilkumaran and R. Rajesh, "Edge detection techniques for image segmentation–a survey of soft computing approaches," *International journal of recent trends in engineering*, vol. 1, no. 2, pp. 250–254, 2009.

[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[25] D. Ciresan, A. Giusti, L. M. Gambardella, and J. Schmidhuber, "Deep neural networks segment neuronal membranes in electron microscopy images," in *Advances in neural information processing systems*, pp. 2843–2851, 2012.

[26] *Stanford University Computer Science Course CS231n preprocessing.* `http://cs231n.github.io/neural-networks-2/`. Accessed: 2017-08-07.

[27] T. Huang, G. Yang, and G. Tang, "A fast two-dimensional median filtering algorithm," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 27, no. 1, pp. 13–18, 1979.

[28] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pp. 234–241, Springer, 2015.

[29] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[30] J. W. Johnston, "Similarity indices i: what do they measure.," tech. rep., Battelle Pacific Northwest Labs., Richland, WA (USA), 1976.

[31] Ö. Çiçek, A. Abdulkadir, S. S. Lienkamp, T. Brox, and O. Ronneberger, "3d u-net: learning dense volumetric segmentation from sparse annotation," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pp. 424–432, Springer, 2016.

[32] A. Buades, B. Coll, and J.-M. Morel, "A review of image denoising algorithms, with a new one," *Multiscale Modeling & Simulation*, vol. 4, no. 2, pp. 490–530, 2005.

[33] C. V. Angelino, E. Debreuve, and M. Barlaud, "Patch confidence k-nearest neighbors denoising," in *IEEE ICIP*, 2010.

[34] A. Buades, B. Coll, and J.-M. Morel, "A non-local algorithm for image denoising," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005.*, vol. 2, pp. 60–65, 2005.

[35] G. Chen, T. D. Bui, and A. Krzyzak, "Image denoising using neighbouring wavelet coefficients," *Integrated Computer-Aided Engineering*, vol. 12, no. 1, pp. 99–107, 2005.

[36] J. Xie, L. Xu, and E. Chen, "Image denoising and inpainting with deep neural networks," in *Advances in Neural Information Processing Systems*, pp. 341–349, 2012.

[37] L. Gondara, "Medical image denoising using convolutional denoising autoencoders," in *Data Mining Workshops (ICDMW)*, pp. 241–246, 2016.

[38] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[39] A. Kessy, A. Lewin, and K. Strimmer, "Optimal whitening and decorrelation," *The American Statistician*, to appear 2017.

[40] P. Felkel, R. Wegenkittl, and A. Kanitsar, "Vessel tracking in peripheral cta datasets-an overview," in *Computer Graphics, Spring Conference on, 2001.*, pp. 232–239, IEEE, 2001.

[41] *Notes from Russel Stewart on NaNs*. http://russellsstewart.com/notes/0.html. Accessed: 2017-08-07.

[42] S. Perreault and P. Hébert, "Median filtering in constant time," *IEEE transactions on image processing*, vol. 16, no. 9, pp. 2389–2394, 2007.