Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Chair for Mathematical Information Science
Prof. Dr. H. Bölcskei

Bachelor Thesis in Mathematics

Fall Semester 2021

Pablo Lahmann

Stability of Simple Neural Network Architectures

Supervisor: Thomas Allard

December 2021

Abstract

Neural networks have become a powerful tool to handle many analytical problems. This thesis explores the performance and stability of simple neural networks and serves as an introduction into programming neural networks with Python.

Throughout this report, we will also focus on explaining the functions and strategies that were employed when writing the code.

We start out by sketching some of the theory, which is relevant for this report. Then we program a simple neural network and use it to interpolate the data from a linear function with noise. Later, we make some observations, which we discuss in the following sections. After finishing this, we examine slightly more complex neural networks and compare their performances. This will allow us to make deductions about what kinds of neural network architectures are preferable to others.
In a next step, we analyze the stability of a simple neural network architecture in two different ways. First, we derive analytical solutions which fit the generated data perfectly. This gives us some constraints, which the weights fulfill, if they have minimal loss. Depending on the degree to which these constraints are fulfilled, we can make assumptions about the stability of the optimized neural network. Since we cannot always derive all analytical solutions to a neural network to fit data, we also use a second method to analyze the stability of optimized weights. This method uses contour plots, which chart the loss for a variety of possible weight-configurations. Depending on how the area of values for weights, which have minimal loss, looks like, we can get a sense of how stable the neural network is. We perform this analysis for several neural network architectures and also for other functions.

Lastly we examine how variations of the learning rate affect the stability of a simple neural network structure and observe, that for larger learning rates, stochastic gradient descent (SGD) exhibits a tendency to optimize towards specific configurations of weights.

Acknowledgments

I thank my supervisor Thomas Allard for helping me out when I was stuck and helping me to adjust my focus on several occasions.
In addition I owe gratitude to Prof. Dr. H. Bölcskei, who gave me the opportunity to work on this project.

Contents

Chapter 1

Underlying Theory

The first chapter of this report introduces key topics which are relevant for the coming discussions. Particularly, the concept of artificial neural networks and activation functions, which are used to identify patterns in data and using this information to extrapolate. We will present the ReLU activation function. Lastly, we describe Stochastic Gradient Descent (SGD), which is an optimization algorithm used to adapt a neural network to given data. The information provided in this chapter relies on the explanations in [1] ($p. 149 - 150$ and $p. 164 - 173$).

1.1 Neural Networks and Activation Functions

This report focuses on (feedforward) neural networks. Such networks describe certain kinds of functions, which can be used to fit a set of data points. Here we will only look at data which consists of one-dimensional inputs in $\mathbb{R}$ each of which is associated with a one-dimensional output in $\mathbb{R}$. Such neural networks can be represented with directed acyclic graphs. An example of such a graph is illustrated in Figure 1.1.

Furthermore, we can group all nodes into sets of "layers". For every layer of nodes, there exists a subsequent layer of nodes, and all nodes from the first layer share an edge to every node in the following layer (the edge is always directed towards the node in the subsequent layer). In a more theoretical sense: if all edges were undirected, each subgraph consisting only of two adjacent layers is a complete bipartite graph. We call such neural networks dense. This form can be seen in Figure 1.1, where each column of nodes is a layer.

In a neural network graph, we call the first layer the input layer and the last, the output layer. All other layers are commonly referred to as hidden layers. Every node in a layer corresponds to a value, which is computed in the process of computing the output value of the neural network. As the name implies, the output node corresponds to the output value of the neural network, while the input node represents an input value. Additionally, the edges represent weights, which are used to calculate the value for each node. To be more specific, if in Figure 1.1 the input node corresponds to an input x, then to calculate the value for node 1 we multiply the input with the respective

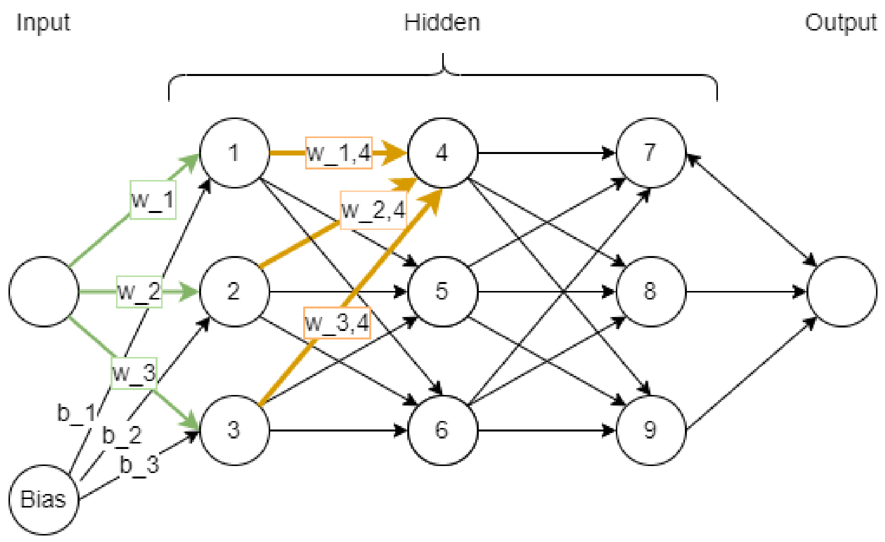


Figure 1.1: This graph is used to describe a neural network. It has exactly one node in the input layer and in the output layer, which corresponds to the case that both the input and the output of this neural network are one-dimensional. Each edge corresponds to an adjustable weight of the neural network.

weight i.e. w_1x. Furthermore, it is possible to add a bias node to a layer. This means that to calculate the value of a node in the subsequent layer, we add a node-specific constant weight. In the case of Figure 1.1, we add a bias node to the input layer. Thus the value of node 1, which is part of the subsequent first hidden layer, is given by $w_1x + b_1$. This corresponds to the bias node having the value 1.
But calculating the values of the nodes in other layers is slightly different. To illustrate this, we give an example of how to calculate the values of a node in the second hidden layer of Figure 1.1. First note that we write v_i for the value if the ith node. In that case it holds that for an input x and a function $f_1 : \mathbb{R} \to \mathbb{R}$, the value of the 4th node is given by

$$\begin{aligned} v_4 &= w_{1,4} \cdot f_1(v_1) + w_{2,4} \cdot f_1(v_2) + w_{3,4} \cdot f_1(v_3) \\ &= w_{1,4} \cdot f_1(w_1x + b_1) + w_{2,4} \cdot f_1(w_2x + b_2) + w_{3,4} \cdot f_1(w_3x + b_3). \end{aligned}$$

It is important to note, that when calculating the value of a node in the neural network (except for n node which is part of the first hidden layer), we multiply the weights w with $f_i(v)$, where v is the value of the respective node in the previous layer. We call such a function f_i an activation function. These are necessary, because otherwise the resulting composition of functions to receive the value of the output node would simply be a linear function in terms of the input value. But clearly, this is too restricting, especially when trying to fit nonlinear patterns in data. This is why activation functions are used to introduce some nonlinearity into its interpolating capabilities. The Rectified Linear Unit (short: ReLU) a widely used activation function, which we will be analyzing more closely in the following chapters. It is defined by $ReLU(x) = \max\{0, x\}$ and thus consists of two linear parts, one of which is constantly zero. The large similarities to linear functions combined with some nonlinearity make the $ReLU$ activation function a very powerful tool.

1.2 Mean Squared Error and Stochastic Gradient Descent

SGD is an optimization algorithm, which is used when working with neural networks. It is tasked with finding good values for the weights, such that the given neural network can fit specific data accurately. To use such algorithms, it is necessary to have a means of quantifying how well a given neural network is extrapolating the data. Precisely this is the purpose of loss functions. In particular, in this report we will discuss the loss function of the Mean Squared Error (MSE).
Let our data consist of N points (x_i, y_i) for $i \in \{1, 2, ..., N\}$ and let $N_\theta(x)$ be the output of a neural network with some given set of weights θ, then the MSE is given by

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - N_\theta(x_i))^2.$$

Thus the goal of SGD is to find weights which minimize this MSE.
The method of SGD is based on gradient descent, in which we regard the MSE as a function of the weights θ. For some data set of points (also called: training set), we minimize the $MSE(\theta)$, by initializing θ with some initial value θ_{init} and computing the corresponding loss. Starting at the value $MSE(\theta_{init})$, we iteratively update the estimate of a value θ_{min} for which the MSE is minimal. This happens by replacing the old estimate θ_{old} with the value

$$\theta_{new} \leftarrow \theta_{old} - \eta \cdot \nabla_\theta MSE(\theta_{old})$$

for some learning rate $\eta \in \mathbb{R}$. Note that $-\nabla_\theta MSE(\theta_{old})$ is the direction of the steepest descent at the point θ_{old} in the values of $MSE(\theta)$. Thus after one iteration in gradient descent, we moved our new estimate θ_{new} into a direction of lower loss.
A problem with gradient descent is that as the number of points in our training set increases, so does the computational cost of calculating the MSE. SGD solves this problem by only computing an approximation of the MSE for some given weights θ. To achieve this, the optimization algorithm chooses uniformly a small subset (also called: minibatch) of m points from the training data. Only these points are used to compute the MSE. Specifically in the way that at each iteration we choose uniformly some $k_1, ..., k_m \in \{1, ..., N\}$ where usually $m \ll N$. Then the following update is made:

$$\theta_{new} \leftarrow \theta_{old} - \eta \cdot \nabla_\theta \left(\frac{1}{m}\sum_{i=1}^{m}(y_{k_i} - N_\theta(x_{k_i}))^2 \right).$$

After a predetermined number of iterations, SGD terminates and receives a candidate for θ_{min}.

Chapter 2

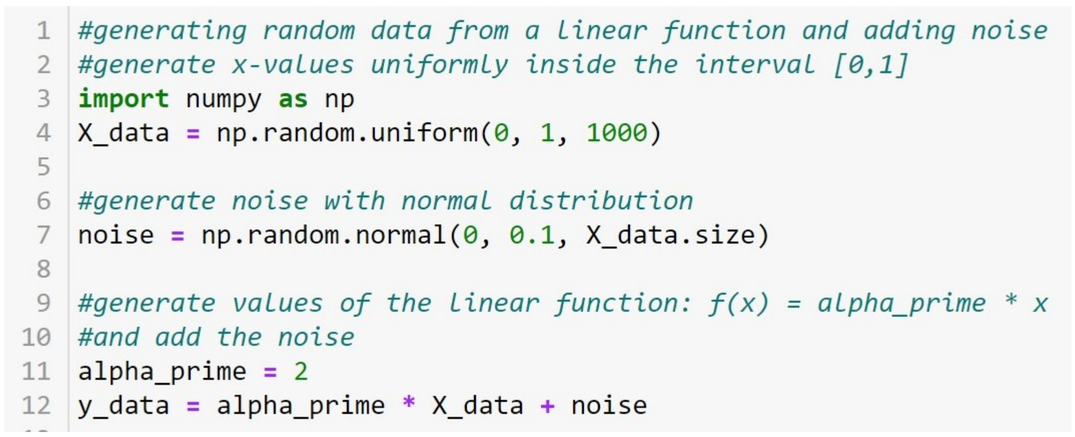
Programming a simple neural network

In this chapter we will gain some hands-on experience in programming a neural network and optimizing it. We will use the Python libraries TensorFlow and Keras. On every step of the way, we explain which TensorFlow-specific functions are needed to import and which commands should be used.
First, we construct a neural network, which has a very simple architecture (with ReLU activation) and use it to interpolate data generated from a linear function with additive noise. In the process of optimization, we notice that the Mean Squared Errors of well optimized neural networks frequently take on very similar values. We will explain this minimal error in more details. Furthermore, we notice that our neural network, with ReLU-activated nodes, surprisingly often fails to optimize sufficiently. We will describe this problem and analyze what it tells us about the advantages and disadvantages of ReLU-activated nodes in a neural network.
In the following chapters we will then gradually increase the complexity of the neural networks and the functions, which we interpolate.
The code presented here and in the subsequent sections can be found in [2].

2.1 Interpolating a linear function with noise

Our first goal is to interpolate the data from a linear function. For this, we will look at $f(x) = \alpha^* x$, for some $\alpha^* \in \mathbb{R}$ and $\alpha^* > 0$, on the interval $[0, 1]$. For this we will generate some data, on which we will optimize the neural network. The detailed code for implementing this can be seen in Figure 2.1. Here, we first need to import the Python library NumPy to use its functions for generating random points on some interval. Specifically, we want to generate 1000 random uniformly distributed points inside the interval $[0, 1]$.
As our next step, we generate the noise, which we then add to our data. The noise is generated with normal distribution around the mean $\mu = 0$ with a standard deviation of $\sigma = 0.1$. For this we use the function

```
numpy.random.normal(m,s,n),
```



```python
#generating random data from a linear function and adding noise
#generate x-values uniformly inside the interval [0,1]
import numpy as np
X_data = np.random.uniform(0, 1, 1000)

#generate noise with normal distribution
noise = np.random.normal(0, 0.1, X_data.size)

#generate values of the linear function: f(x) = alpha_prime * x
#and add the noise
alpha_prime = 2
y_data = alpha_prime * X_data + noise
```

Figure 2.1: Program 1 [2] - Generating the data to be interpolated.

Which returns an array of n points with normal distribution. Where m $= \mu$, the mean and s $= \sigma$, the standard deviation.
The variable `alpha_prime` is equal to the slope α^* of our linear function $f(x)$. For our computations we will fix $\alpha^* = 2$. If we want to do all computations with a different slope, we can simply change the value which is assigned to `alpha_prime` in the beginning of each program.
Finally, we generate the data which will be interpolated. It is given by the values, which $f(x)$ attains at the positions in `X_data` plus the noise.

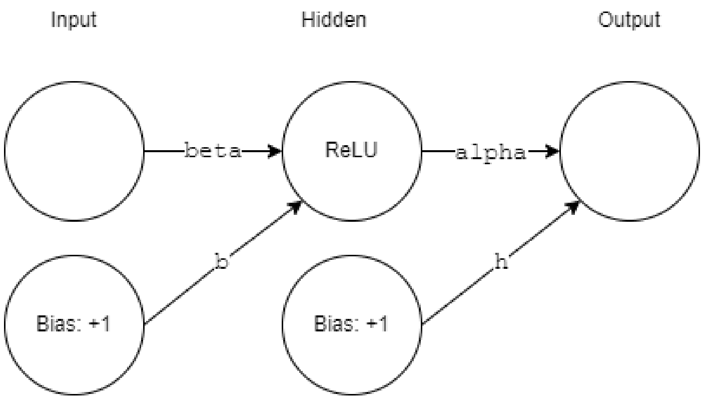
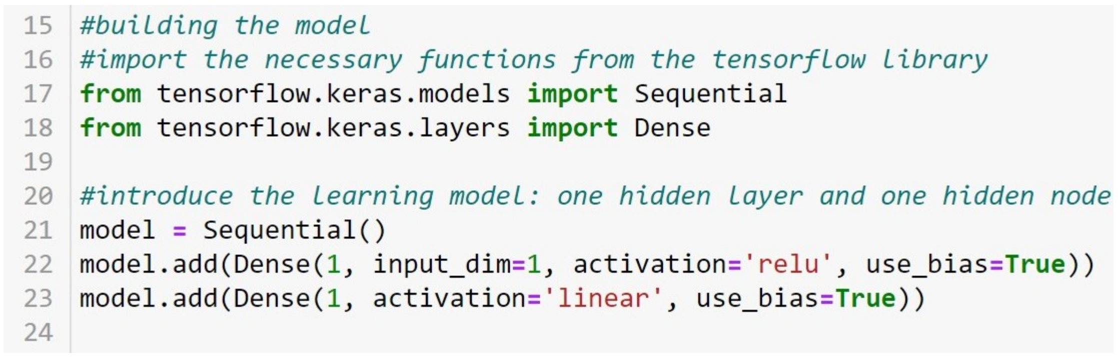


Figure 2.2: Simple neural network with one ReLU-activated node inside exactly one hidden layer. The markings on the arrows correspond to the weights in the neural network.
In the rest of the report we will refer to this architecture as the "**Simple NN**".

Now that we have some data, we can get started on constructing our neural network. We will focus on a very simple neural network architecture with one hidden layer containing exactly one hidden node as well as bias nodes in the input layer and hidden layer (see Figure 2.2).



```
15 #building the model
16 #import the necessary functions from the tensorflow library
17 from tensorflow.keras.models import Sequential
18 from tensorflow.keras.layers import Dense
19
20 #introduce the learning model: one hidden layer and one hidden node
21 model = Sequential()
22 model.add(Dense(1, input_dim=1, activation='relu', use_bias=True))
23 model.add(Dense(1, activation='linear', use_bias=True))
24
```

Figure 2.3: Program 1 [2] - Implement the neural network as described in Figure 2.2.

In the following we will explain how to build the Simple NN using the Python library TensorFlow. First, we import the sequential function, which allows us to sequentially add new layers to a neural network model (see Figure 2.3). We do this by initializing a model like in line 24 in Figure 2.3. We can now add new layers to our model using the function

```
model.add(Dense(n, activation='func', use_bias=bool)),
```

where `n` is the numbers of nodes inside the layer which we are adding. `func` is a placeholder for an abbreviation for pre programmed activation functions. In our example, we use a ReLU activated node (`func` = `relu`) in our hidden layer and a linearly activated node (`func` = `linear`) in our output layer. We can decide if a bias node should be included, by writing either `True` or `False` instead of `bool`. Note that in line 25 we also write `input_dim = 1`. This is a necessary thing to write when adding the first layer to our model, to clarify the dimension of the input. In our case of interpolating a linear function, the input values will consist of one-dimensional x-values inside our interval $[0, 1]$. Additionally, we import and use the function `Dense` which is applied to build fully connected layers. This means that all nodes in two adjacent layers of our model are interconnected. As we can see in Figure 2.2 this is clearly the case in the Simple NN.

We remark that the Simple NN as in Figure 2.2 is essentially just a way of describing an interpolating function. In our case the Simple NN is equivalent to the function

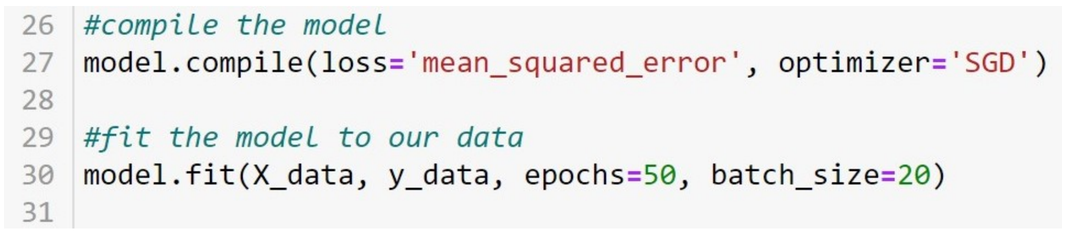
$$N(x) = \alpha ReLU(\beta x + b) + h,$$

where the variables α, β, b, h are the weights of the Simple NN as in Figure 2.2.

Now we will look further into how we find good values for α, β, b, h, for which the MSE is small. In Figure 2.3 we have set up the architecture of the Simple NN which we want to use. As we can see in Figure 2.4 in our next step we configure the model with losses and metrics using the function

```
model.compile(loss='mean_squared_error', optimizer='SGD'),
```

where `loss` determines which loss function we want to minimize when fitting the neural network. In our case we decide to use the Mean Squared Error, which we



```
#compile the model
model.compile(loss='mean_squared_error', optimizer='SGD')

#fit the model to our data
model.fit(X_data, y_data, epochs=50, batch_size=20)

```

Figure 2.4: Program 1 [2] - Compiling and fitting the Simple NN to our data.

have introduced in Section 1.2. The attribute `optimizer` tells the program which optimization algorithm should be used to minimize the loss function on the training data. In this project we focus on the optimization algorithm of Stochastic Gradient Descent (SGD), of which we have seen the theory in Section 1.2.
Lastly, in Figure 2.4, we use the function

```
model.fit(X_data, y_data, epochs=E, batch_size=B)
```

to fit the weights in our model to the data we have generated before. The training set is given by `X_data`, the input, and `y_data`, the desired output. On this data we want to train our neural network. The number of epochs `E` fixes the number of iterations we want to make in our optimization. The batch size `B` determines how large the subset of data points on which we employ the iteration is. In our case of SGD, it represents the number of points which we use to compute the gradient.

```
Epoch 50/50
50/50 [==============================] - 0s 855us/step - loss: 0.0112
```

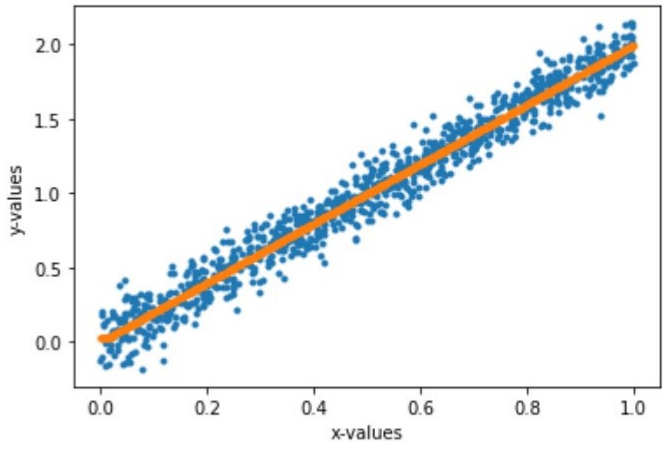
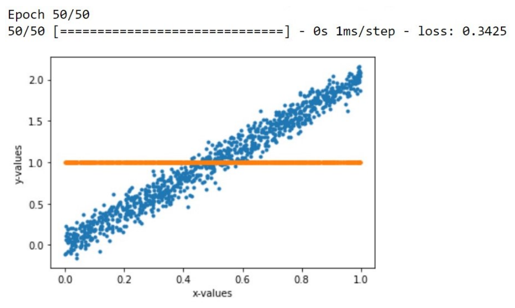


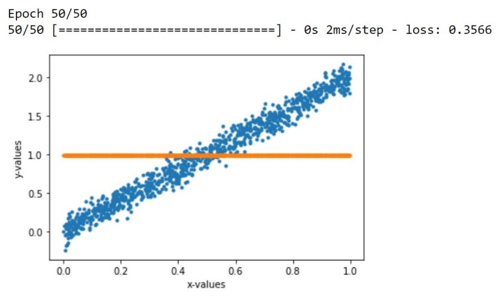
Figure 2.5: Plotting the performance of the Simple NN. On the top part of this figure we can see that the loss of the weights after the last iteration is 0.0112. The orange line represents the predictions of the Simple NN for every value in our data set `X_data`. The blue dots are the original data, which we have generated in the beginning of this section.

Now that we have fit the Simple NN to our data, we can assess how well the opti-



```
Epoch 50/50
50/50 [==============================] - 0s 1ms/step - loss: 0.3425
```

(a) Example 1



```
Epoch 50/50
50/50 [==============================] - 0s 2ms/step - loss: 0.3566
```

(b) Example 2

Figure 2.6: Plotting the performance of the Simple NN. Two examples of bad fittings of our Simple NN to the generated data.

mization went. We will look at two ways of assessing this. First, by looking at the loss of our neural network after the last epoch of our optimization. Second, by plotting the predictions of the Simple NN for every point in our data set and also plotting the initial data. An example of these can be seen in Figure 2.5.

After running the code several times, we can see that in the cases, where, judging by the plot, the graph fits the data very well, the loss of the Simple NN after the last iteration is always very close to the value 0.01. This observation is clearly true for Figure 2.5. We will discuss the reason for this in more detail in Section 2.2. In addition, we notice, that quite often during optimization the Simple NN does not fit our data well. In fact, when compiling the program 10 times, the Simple NN fit the data well (i.e. like in Figure 2.5) only 3 times. Two results of these other 7 compilations can be found in Figure 2.6. All of them optimize to a horizontal line on the Interval $[0, 1]$ and to the local minimum of the loss at about 0.35. This happens because the ReLU-node is not activated during the optimization. We will discuss why this happens in more detail in Section 2.3.

2.2 Explanation for common minimal loss in Simple NN

After fitting the Simple NN in Section 2.1, we noticed that the best minimal loss we achieved was always very close to the value 0.01. To explain this, we remind ourselves that the Mean Squared Error is used to give an intuition about how well our neural network fits a curve. More precisely, let $N_\theta(X)$ be the output of our Simple NN, where $\theta = (\alpha, \beta, b, h)$ describes the values of the weights when making a prediction. X describes the input, with the distribution $X \sim \mathcal{U}([0, 1])$. Let $Y(X) = \alpha^* X + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma^2)$ is the noise, which we generated with normal distribution.
Then the Mean Squared Error (MSE) is given by

$$\mathbb{E}_{Y,X}[(Y - N_\theta(X))^2] = \mathbb{E}_{Y,X}[(\alpha^* X + \epsilon - \alpha^* X)^2] = \mathbb{E}_{Y,X}[\epsilon^2] = Var_{Y,X}[\epsilon] = \sigma^2,$$

when assuming that the Simple NN $N_\theta(X)$ approximates the linear function $f(x) = \alpha^* x$ perfectly. We will discuss in Section 3.2 in more detail that this is possible. So we receive that the MSE is exactly equal to the variance of the noise. Furthermore, we generated our noise in Section 2.1 with a standard deviation of $\sigma = 0.1$. Thus the minimal loss of $N_\theta(X)$ must be $\sigma^2 = 0.1^2 = 0.01$, which is very close to what we observed.
We also saw that the minimal loss after optimization was not always exactly equal to 0.01. The reason for this is that after the optimization we are not guaranteed to have converged to the exact solution. Another reason is that above we have only taken the Expected Risk into consideration. Whereas the result from the program gives us in fact the Empirical Risk, which for N data points is given by

$$\frac{1}{N}\sum_{i=1}^{N}(y_i - N_\theta(x_i))^2 = \frac{1}{1000}\sum_{i=1}^{1000}\epsilon_i^2 \approx 0.01,$$

where ϵ_i is the noise which we add to our i-th data point. But note that for a large number of examples the empirical risk approaches the value of the expected risk, by the law of large numbers. That is why the minimal losses in our computations in Section 2.1 are always close to 0.01.

2.3 Things to consider about the ReLU activation

In Section 2.1 we observed that the Simple NN does not optimize well most of the time. The reason for this is the way that the optimization algorithm SGD works. First note that the Simple NN is equivalent to the function $N(x) = \alpha ReLU(\beta x + b) + h$. During the fitting we are looking for values of the weights α, β, b, h which interpolate our data well. As we have seen in Section 1.2, SGD starts by choosing random initialization values of the weights. Notice that it is possible, that for certain initial values, the ReLU inside $N(x)$ does not activate. This means that $\forall x \in [0,1] : ReLU(\beta x + b) = 0$. This is for instance the case then both β and b are negative. Then this means that $N(x) = h$ is a constant function. In these cases the gradient, which we compute for the optimization, will always be zero. In SGD this makes the iteration useless, since it relies on having a non-zero gradient.
This problem of ReLU activation functions can be avoided by initializing the neural network with custom weights for which the ReLU will surely be activated. While we can easily do this for the Simple NN and greatly improve its performance this way, this is not a feasible strategy for more complex neural networks. Since the complexity of a neural network can rise very quickly to an extent, where it is too much work or maybe not even possible to assess all ReLU activated nodes and choose weights which ensure that they are activated.
Therefore we will describe another way of countering the problem, that a ReLU might not be activated at all during optimization. By having many ReLU activated nodes

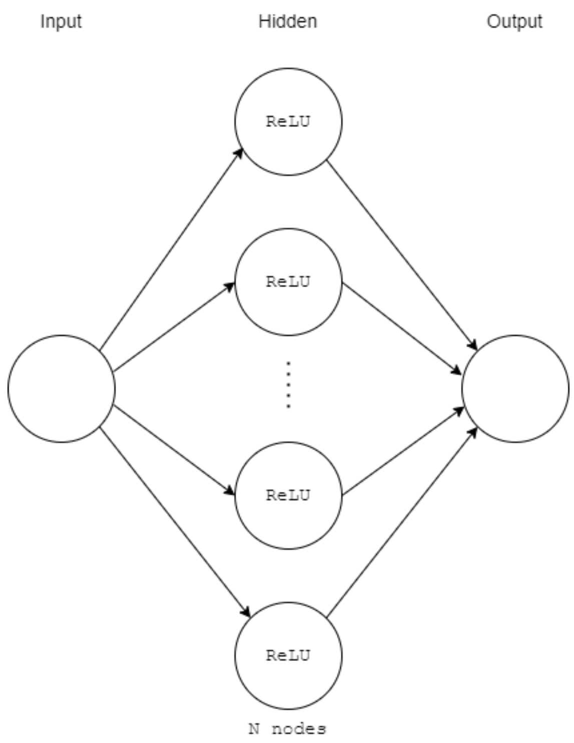


Figure 2.7: Neural network architecture with N ReLU activated nodes in one hidden layer. Note that this neural network is equivalent to the interpolating function $N(x) = \alpha_1 ReLU(\beta_1 x) + ... + \alpha_N ReLU(\beta_N x)$ i.e. a sum of ReLU functions.

in one layer, we decrease the probability that the neural network becomes equivalent to a constant function i.e. that no ReLU node is activated. For this, imagine a neural network which has one hidden layer and N hidden nodes, like in Figure 2.7. This network is equivalent to an interpolating function which consists of a sum of ReLU functions. We only need at least one ReLU node to be activated, to ensure that our neural network has a non-constant output and thus a nonzero gradient. With random initialization of the weights, the probability that at least one node is activated, rises with the number N of nodes inside the layer. Thus it is desirable to have many ReLU activated nodes in one layer.

In contrast, the problem with ReLU activation is amplified when increasing the number of layers with ReLU activated nodes. For this case, we look at the example of the neural network architecture in Figure 2.8, which has N hidden layers and one hidden node inside each of these layers. This network is equivalent to many nested ReLU functions. In this case only one weight needs to be initialized in a way such that its corresponding ReLU function does not activate, such that in turn the entire neural network simplifies to a constant function. From this we can deduce that increasing the amount of layers in a network with ReLU activated nodes, increases the risk that for some initialization values of the weights, several nodes will not be activated and thus not be optimized well or even cause the whole network to optimize badly.

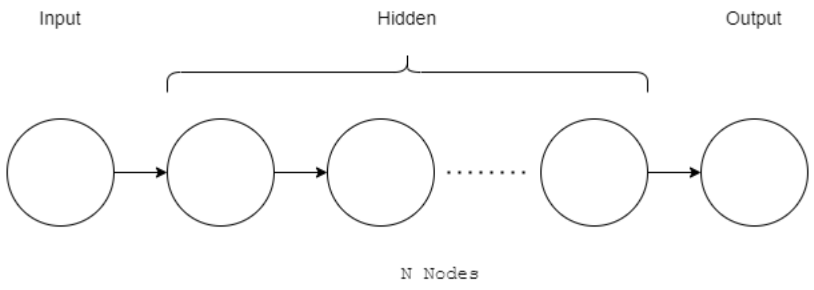


Figure 2.8: Neural network architecture with N ReLU activated nodes in N hidden layers. Note that this neural network is equivalent to the interpolating function $N(x) = \alpha_1 ReLU(\beta_1(...(\alpha_N ReLU(\beta_N x))...))$ i.e. nested ReLU functions.

Consequently it is better to have many ReLU activated nodes in one layer, rather than having many layers with ReLU activated nodes. Adding more layers can degrade the ability to optimize.
At first, this might give the impression that the ReLU activation function is rather impractical. Why not simply use linear activation functions, as they do not fail when initialized with badly chosen weights? But in fact, the non-linearity in ReLU functions is necessary to interpolate the data of non-linear functions. If all nodes in a neural network have linear activation functions, then the entire neural network would be equivalent to a linear interpolating function. Thus, regardless of its disadvantage during optimization, the ReLU activation function is by far more helpful than a linear activation function.

Chapter 3

Important properties for fitting a linear function with noise

In this chapter we will try to test the theory from Section 2.3, which states that having many ReLU activated nodes in one layer is preferred to having them spread out over several layers. Lastly, we will derive some analytical solutions of neural networks for fitting a linear function. We do this in preparation for a stability analysis in the later chapters.

3.1 Other neural network architectures

Remember that in the previous section we concluded that it is advantageous to have many ReLU activated nodes in one hidden layer, rather than having some ReLU activated nodes in many hidden layers. In this section we verify this theory by assessing how often some examples of neural networks optimize well and how often they do not. Particularly, we will look at networks of the same form as in Figure 2.7 and Figure 2.8 for $N = 10$ and with bias in the input node as well as the hidden nodes. Like in the previous chapter, we want these neural networks to interpolate the data from the function $f(x) = \alpha^* x$ on the interval $[0, 1]$ (and we fix $\alpha^* = 2$).
The neural network as in Figure 2.7 fits the linear function optimally with a minimal loss of approximately 0.01 all 10 of the 10 tries. Here we used Program 2 [2]. On the other hand, the neural network as in Figure 2.8, does not optimize below ~ 0.35 in all 10 compilations (See Program 3 [2]).
While this supports our theory, we will continue testing it for two more neural network architectures which also have a total of approximately 10 hidden nodes. These architectures are illustrated in Figure 3.1 and Figure 3.2. As we have done it for the previous architectures, we will also run the fitting process for both architectures ten times. Out of these ten times the architecture in Figure 3.1 optimized well to a loss of approximately 0.01 only two times (see Program 4 [2]). In the other cases, the loss did not get below a value of ~ 0.35. This poor performance is an expected observation, since the neural network has a high amount of layers in comparison to its number of nodes. The neural network with architecture like in Figure 3.2 optimized well six out of ten times (see Program 5 [2]). This supports our theory that as the amount of layers

and nodes in a neural network increase, so does the percentage of times in which the neural network optimizes well to our data.

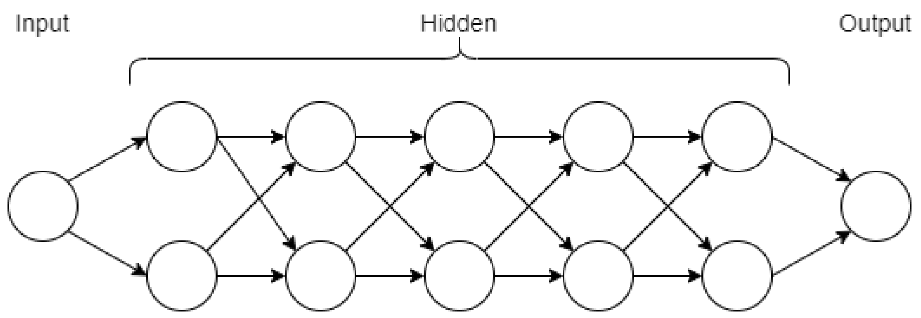


Figure 3.1: Neural network architecture with 5 hidden layers and two ReLU activated nodes in each layer. This is an example of a neural network with a rather large amount of layers with ReLU activated nodes. Thus we expect this neural network to poorly optimize most of the time. (Please note that we program these neural networks with a bias node in the input layer and in every hidden layer. This is not illustrated here.)

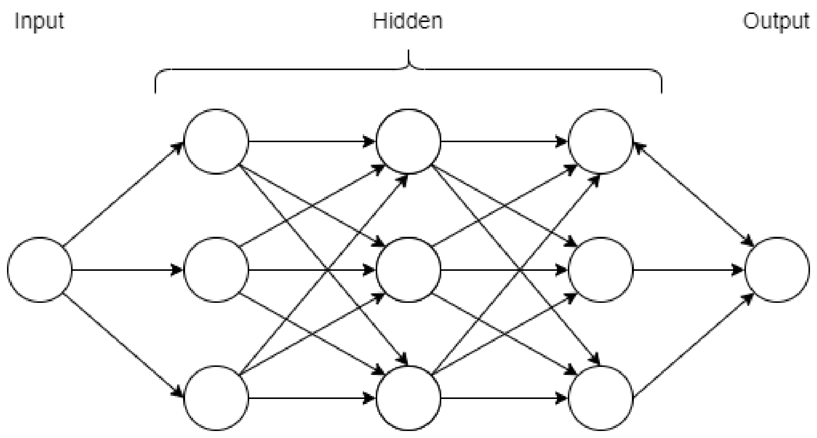


Figure 3.2: Neural network architecture with 3 hidden layers and 3 hidden nodes inside each layer. Since this neural network has fewer layers and more nodes inside each layer, than the neural network in Figure 3.1, we expect it to optimize well, more often. (Please note that we program these neural networks with a bias node in the input layer and in every hidden layer. This is not illustrated here.)

3.2 Deriving constraints for the Simple NN

In this section, we make a preparatory analysis of the ReLU function to, in a later section, better assess the stability of an optimized solution of the weights. In particular, we will be concerned with the Simple NN and all possible weights which solve the optimization problem in Section 2.1 perfectly. We remind ourselves that the Simple

NN is equivalent to the interpolating function

$$N(x) = \alpha ReLU(\beta x + b) + h.$$

This means that we want to find all possible values for $\alpha, \beta, b, h \in \mathbb{R}$ with $\alpha, \beta \neq 0$ (because in these cases the fitting is clearly not fulfilled), which perfectly interpolate the function $f(x) = \alpha^* x$ for $x \in [0, 1]$ and $\alpha^* > 0$.
First note that the ReLU function is given by

$$ReLU(x) = \begin{cases} x, & x > 0, \\ 0, & otherwise. \end{cases}$$

This implies that the ReLU function consists of two linear parts, one of which is a horizontal line for all x-values for which $\beta x + b \leq 0$. At all other x-values the ReLU function behaves like a second arbitrary linear function, we call this the inclined part of the ReLU function. This is illustrated in Figure 3.3. Further, depending on the sign of α, the inclined part of $N(x)$ can tilt upwards or downwards. The sign of β controls whether the horizontal part of $N(x)$ is to the left or to the right of the inclined part of the ReLU. An illustration of these two relations can be found in Figure 3.3.
In the cases (1) and (4) of Figure 3.3 it holds that $\beta > 0$, because the horizontal part of $N(x)$ is to the left of the inclined part. Which means that there must exist some threshold $t \in \mathbb{R}$, such that the ReLU function $ReLU(\beta x + b) = 0$ for all $x < t$ (which implies that $N(x)$ is constant). This is equivalent to stating that $\beta x + b < 0$ for all $x < t$, and we know that a linear function can only fulfill this condition, if its slope β is positive. We can argue similarly, to justify that in the cases (2) and (3) we must have that $\beta < 0$.
Note that when we write $ReLU(\beta x + b)$, we modify the linear function $\beta x + b$ in the way all x-values for which the linear function attains negative values, are instead sent to 0. Thus only the positive values which $\beta x + b$ attains are left unchanged by the ReLU function. This means that regardless of the values of β and b, the term $ReLU(\beta x + b)$ will always have an inclined part which tilts upwards (when starting out at the horizontal part). Multiplying $ReLU(\beta x + b)$ with some positive α will only change the position of the graph and the steepness of the inclined part, but the inclined part will still tilt upwards (like in the cases (1) and (2) in Figure 3.3). If on the other hand, α is negative, then $\alpha ReLU(\beta x + b)$ will clearly have a inclined part which tilts downwards, like in the cases (3) and (4) in Figure 3.3.

Since we want to approximate a linear and inclined function (with positive slope) on the interval $[0, 1]$, we need $N(x)$ to be inclined on this interval. As a consequence, the horizontal part of the ReLU function must either be to the left of the interval $[0, 1]$, like in Figure 3.4, or the horizontal part must be to the right of the interval, like in Figure 3.5. These two cases can be characterized by the value of h. If we examine the x-values for which $N(x)$ is a horizontal line, we know that these values fulfill $ReLU(\beta x + b) = 0$, which implies that $N(x) = \alpha \cdot 0 + h = h$ is constant. This means that the horizontal line will always be at the height h. Depending on the sign of α, $N(x)$ will not attain any values below (if $\alpha > 0$) or above (if $\alpha < 0$) the value of h (see Figure 3.3 to visualize this). But inside our interval the interpolated function takes all values from

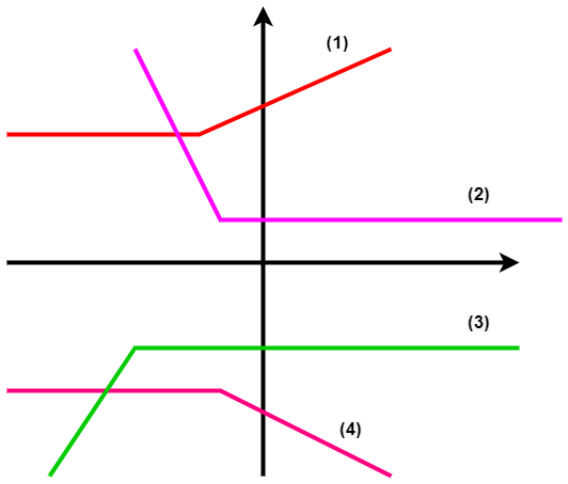


Figure 3.3: $N(x) = \alpha ReLu(\beta x + b) + h$ has four different ways it can look in a plot, depending on whether the inclined part tilts up- or downwards and whether the horizontal part is left or right of the inclined part. Case (1): $\beta, \alpha > 0$. Case (2): $\beta < 0$, $\alpha > 0$. Case (3): $\beta, \alpha < 0$. Case (4): $\beta > 0$, $\alpha < 0$.

$[0, \alpha^*]$. Thus either $h \leq 0$ and $\alpha > 0$ (and $\beta > 0$; See Figure 3.4) **or** $h \geq \alpha^*$ and $\alpha < 0$ (and $\beta < 0$; See Figure 3.5). Therefore we will divide the derivation of the analytical solution of the interpolation into two cases. But first, we will do some general analysis of the interpolation problem.

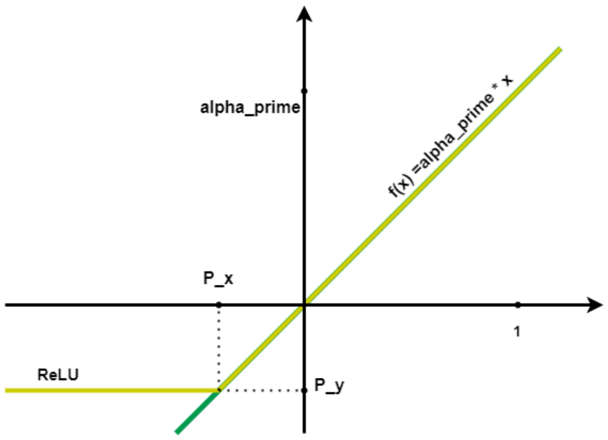


Figure 3.4: The horizontal part of $N(x)$ is to the left of the interval $[0, 1]$ and the linear part tilts upwards, i.e. $h \leq 0$, $\alpha > 0$ and $\beta > 0$.

First we define the point $P = (P_x, P_y)$ to be the breaking point of the function $N(x)$. This is the point, where $N(x)$ transitions from being inclined to being horizontal. Since $N(x)$ is horizontal when $\beta x + b < 0$ and inclined when $\beta x + b > 0$, it clearly holds that the breaking point is exactly where $\beta x + b = 0$. This means that we can write $\beta \cdot P_x + b = 0 \Rightarrow P_x = -\frac{b}{\beta}$. Furthermore we have already established that on its horizontal part, $N(x) = h$. Thus $P_y = h$ must be true (we recommend looking at Figure 3.4 to visualize the reason for this).

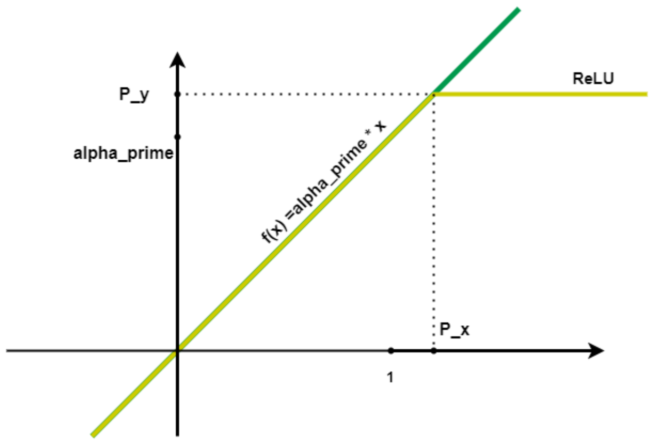


Figure 3.5: The horizontal part of $N(x)$ is to the right of the interval $[0, 1]$ and the inclined part tilts downwards, i.e. $h \leq 0$, $\alpha > 0$ and $\beta > 0$.

Further note that the inclined part of $N(x)$ needs to have the same slope as $f(x) = \alpha^* x$, i.e. the slope α^*. We know that for all x-values on which $N(x)$ is inclined, the ReLU function inside $N(x)$ is activated. Thus we have that $ReLU(\beta x + b) = \beta x + b$, which is why $N(x)$ evaluates to $N(x) = \alpha ReLU(\beta x + b) + h = \alpha(\beta x + b) + h = \alpha\beta x + \alpha b + h$. Such that the slope of $f(x)$ and the inclined part of $N(x)$ are the same, the equality $\alpha^* = \alpha\beta$ must hold true. This gives us a first constraint for the weights of the Simple NN, to fit $f(x)$ perfectly.
In the next part we address two separate cases:
Remember that $N(x) = \alpha ReLU(\beta x + b) + h$.

Case 1: $h \leq 0$, $\alpha > 0$ and $\beta > 0$. This is illustrated in Figure 3.4.
Note that since the horizontal part of $N(x)$ is to the left of the interval $[0, 1]$, so must be the breaking point $P = (P_x, P_y)$. Thus $-\frac{b}{\beta} = P_x \leq 0$ must hold. Also, the breaking point P must lie on the graph of the function $f(x) = \alpha^* x$, which is why another condition is given by $h = P_y = \alpha^* \cdot P_x = \alpha^* \cdot \left(-\frac{b}{\beta}\right)$ (this follows from the analysis we have done above). In total, we receive the following three conditions which must be fulfilled by the weights, such that $N(x)$ interpolates the data from $f(x)$ perfectly:

$$\text{(i)} \quad -\frac{b}{\beta} \leq 0 \qquad \text{(ii)} \quad \alpha^* = \alpha\beta \qquad \text{(iii)} \quad h = \alpha^* \cdot \left(-\frac{b}{\beta}\right).$$

Simplifying these constraints by multiplying equation (i) with -1 and inserting equation (ii) into equation (iii), gives us the final constraints, which $N(x)$ has to fulfill, to fit $f(x)$ perfectly like in Figure 3.4:

$$\text{(i)} \quad \frac{b}{\beta} \geq 0 \qquad \text{(ii)} \quad \alpha^* = \alpha\beta \qquad \text{(iii)} \quad h = -\alpha b.$$

Case 2: $h \geq \alpha^*$, $\alpha < 0$ and $\beta < 0$. This is illustrated in Figure 3.5.
Following the same reasoning as in Case 1, we deduce that $-\frac{b}{\beta} = P_x \geq 1$ as well as $\alpha\beta = \alpha^*$ must be true. Since, like in Case 1, the breaking point P has to lie on the graph

of $f(x) = \alpha^* x$, the equation $h = \alpha^* \cdot \left(-\frac{b}{\beta}\right)$ must be true. In total, this gives us three constraints, which are very similar to the ones in Case 1:

(i) $-\frac{b}{\beta} \geq 1$ (ii) $\alpha^* = \alpha\beta$ (iii) $h = \alpha^* \cdot \left(-\frac{b}{\beta}\right)$.

Simplifying these constraints by multiplying equation (i) with -1 and inserting equation (ii) into equation (iii), gives us the final constraints, which $N(x)$ has to fulfill to fit $f(x)$ perfectly like in Figure 3.5:

(i) $\frac{b}{\beta} \leq -1$ (ii) $\alpha^* = \alpha\beta$ (iii) $h = -\alpha b$.

These two cases give us some constraints for when the Simple NN fits the liner function $f(x)$ perfectly. In the following chapter we will use this, to analyze the stability of an optimized Simple NN.

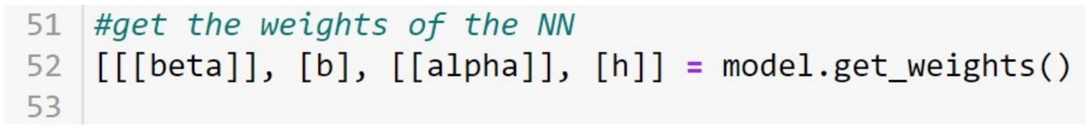
Chapter 4

Stability of the Simple NN

In this chapter we look at different ways of how to assess the stability of our trained Simple NN. For this we use two different ways of analysis. First, we see how well optimized weights fulfill the previously derived constraints. Afterwards, we draw contour plots which illustrate which area of weight-values the loss is very low.

4.1 Assess the stability using the constraints

In this section we perform a first analysis of the stability of the Simple NN after optimization. For this we expand the code from Section 2.1 to first get the weights of the optimized neural network and then check how well the constraints from Section 3.2 are fulfilled. We consider an optimization stable, if even after slightly changing the values of the weights, the constraints are still fulfilled. As an example, we consider the product $1 \cdot 2 = 2$ to be a stable product, while the product $0.0001 \cdot 20000 = 2$ is not a stable product, since a small change in the value 0.0001 results in a significant change of the product.



```python
#get the weights of the NN
[[[beta]], [b], [[alpha]], [h]] = model.get_weights()
```

Figure 4.1: Program 6 [2] - Get the weights of our Simple NN after fitting it to the data. Note that we have to be careful to consider the format in which the weights are output by the function. We receive separate arrays for all the weights in each layer and an array with all bias values.

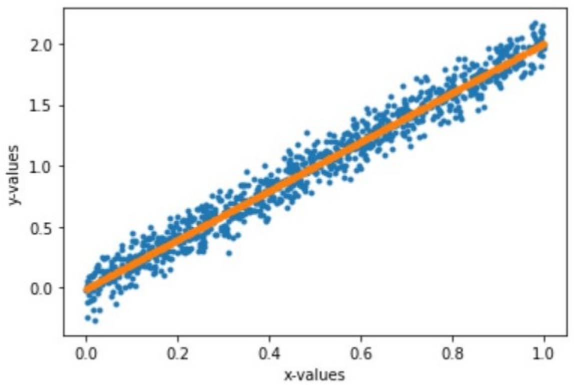
As we can see in Figure 4.1, in Python we use the function

```
model.get_weights()
```

to get the weights of the Simple NN after fitting (see Program 6 [2]).
In a next step, we print all constraints for the optimized weights. To know if we should

check the constraints from Case 1 or Case 2, we use an if-clause to determine which preconditions are fulfilled. Then we print the respective constraints. An example of such a compilation can be found in Figure 4.2. We observe that the constraints are fulfilled in a stable manner. Note that equation (i) is technically not fulfilled, but since we are working with approximations, in the case of Figure 4.2, this is well enough. Note that the term $\frac{b}{\beta}$ is very close to zero, because $b \approx 0$ and β is close to 1. Thus this can be considered a stable product. In equation (ii) the product is approximately $1.80 \cdot 1.12 \approx 2$, which as discussed before is stable. Similarly to equation (i), equation (iii) is stable because one factor in the product is very close to 0.
All other successful optimizations of the Simple NN to the data i.e. with a loss of 0.01, also fulfilled the constraints in a very similar stable manner.

```
Epoch 50/50
50/50 [==============================] - 0s 1ms/step - loss: 0.0107
```

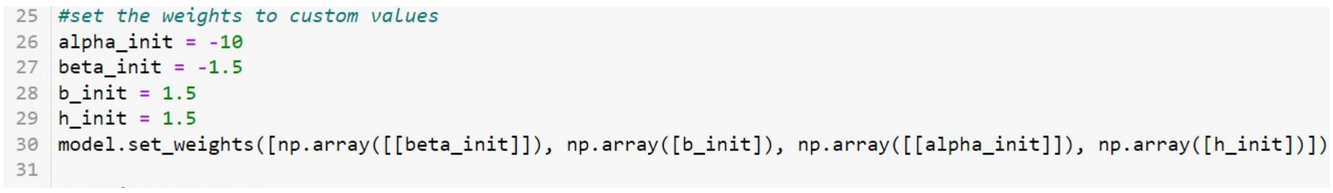


```
Boundary conditions
The horizontal part of the ReLU is left of the interval [0,1].
(i)  b/beta =  -0.0034012122  =  -0.0038014078 / 1.1176627  >=? 0
(ii)  alpha_prime =  2  =?  2.0077207  =  1.7963566 * 1.1176627  = alpha * beta
(iii)  h =  -0.0010199458  =?  0.006828684  = - 1.7963566 * -0.0038014078  = -alpha * b
```

Figure 4.2: After fitting the Simple NN, we get the weights and print them as well as their constraints, to better assess how well the conditions are fulfilled. In this example we can see that the Simple NN fit the data like theorized in Case 1 in Section 3.2. The question marks ? highlight the (in-)equalities which should be fulfilled, according to the constraints.

4.2 Stability at different initial weights

We discussed in Section 2.3 how initializing a neural network with custom weights before executing the optimization is useful to ensure that a ReLU node is activated. In this section we elaborate the possibility of choosing custom initial weights to the Simple NN. Then we will examine how this affects the stability of the optimized weights. Note that in this section we use Program 6 [2].



```python
#set the weights to custom values
alpha_init = -10
beta_init = -1.5
b_init = 1.5
h_init = 1.5
model.set_weights([np.array([[beta_init]]), np.array([b_init]), np.array([[alpha_init]]), np.array([h_init])])

```

Figure 4.3: Program 6 [2] - Choosing custom initial weights of the Simple NN before compiling the model. Like in Figure 4.1 also here we have to careful to consider the format in which the weights should be input. This is taken into account by using the specific array structure in the code.

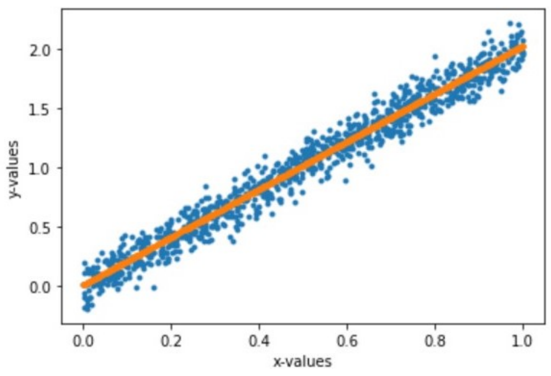
Like in Figure 4.3 we use the function

```
model.set_weights(...)
```

to set custom initial values for the weights.
We know from Section 4.1 that if the weights are not initialized with custom values and the Simple NN optimizes well (to a loss of around 0.01), then the optimized values are stable. In the following we test how different initialization values for the weights affect the stability of the Simple NN. More precisely, we gradually increase the perturbation in the initialized weights and examine how this affects the stability of the optimized weights.
In Figure 4.4, we initialized the Simple NN with perfect solutions, which fulfill the constraints from Section 3.2. In both cases we observe that the optimized weights are close to the initial values, thus they remain stable.
In Figure 4.5, we initialize the Simple NN with an unstable solution and observe that it optimizes very badly, even though the initial values were a perfect analytical solution. This means that we should expect the optimized weights of the neural network to always deviate to some degree from the initial values. A reason for this might be that during the optimization the weights experience some perturbation, which is strongly amplified when initializing the neural network with an unstable solution.
In Figure 4.6, we see how different degrees of perturbed initial weights affect the stability of the optimized Simple NN. Any further increase in perturbation causes the Simple NN to not optimize below a loss of ~ 0.35.

We find that for stable initial weights we receive a neural network, which optimizes well and is stable. Increasing both the instability of an initial solution as well as its perturbation, lead to either poorly fitting of the data or highly unstable solutions.

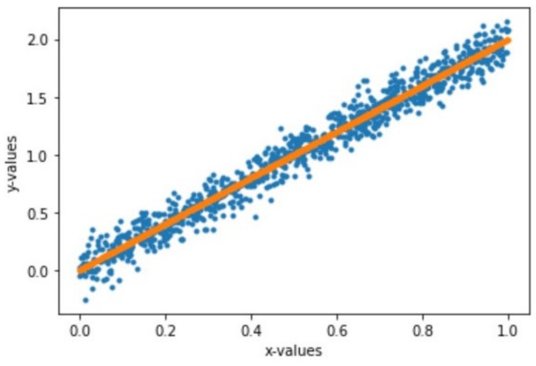
```
Epoch 50/50
50/50 [==============================] - 0s 997us/step - loss: 0.0108
```



```
The Simple NN was initiliazed with the weights: alpha_init =  2  beta_init =  1  b_init =  0  h_init =  0
Constraints:
The horizontal part of the ReLU is left of the interval [0,1].
(i)  b/beta =  0.004810333  =  0.004834913 / 1.0051099  >=? 0
(ii)  alpha_prime =  2  =?  2.01207  =  2.0018408 * 1.0051099  = alpha * beta
(iii)  h =  0.0025003143  =?  -0.009678727  = - 2.0018408 * 0.004834913  = -alpha * b
```

(a) Results of initializing the Simple NN with a perfect solution, which corresponds to Case 1 as in Section 3.2.

```
Epoch 50/50
50/50 [==============================] - 0s 1ms/step - loss: 0.0103
```

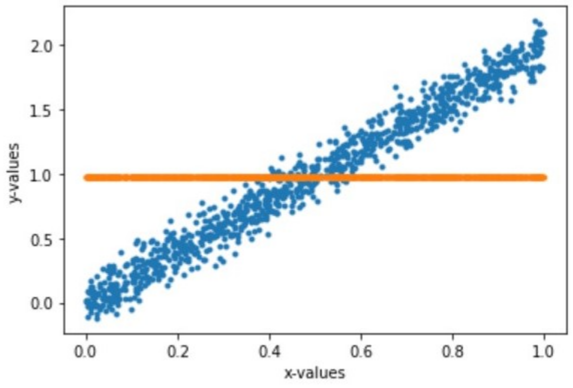


```
The Simple NN was initiliazed with the weights: alpha_init =  -2  beta_init =  -1  b_init =  1  h_init =  2
Constraints:
The horizontal part of the ReLU is right of the interval [0,1].
(i)  b/beta =  -1.0012456  =  0.9991972 / -0.99795413  <=? -1
(ii)  alpha_prime =  2  =?  1.9937999  =  -1.9978874 * -0.99795413  = alpha * beta
(iii)  h =  2.0012448  =?  1.9962834  = - -1.9978874 * 0.9991972  = -alpha * b
```

(b) Results of initializing the Simple NN with a perfect solution, which corresponds to Case 2 as in Section 3.2.

Figure 4.4: Initializing the Simple NN with perfect solutions. We can see that the optimized weights are very stable.

```
Epoch 50/50
50/50 [==============================] - 0s 1ms/step - loss: 0.3407
```



```
The Simple NN was initiliazed with the weights: alpha_init =  0.02  beta_init =  100  b_init =  0  h_init =  0
Constraints:
The Simple NN did not manage to fit the data well.
(i.case1)  b/beta =  2.7025595  =  -351.169 / -129.93942  >=? 0
(i.case2)  b/beta =  2.7025595  =  -351.169 / -129.93942  <=? -1
(ii)  alpha_prime =  2  =?  160362.92  =  -1234.136 * -129.93942  = alpha * beta
(iii)  h =  0.9750562  =?  -433390.3  = - -1234.136 * -351.169  = -alpha * b
```

Figure 4.5: Initializing the Simple NN with an unstable perfect solution. Note that fitting the Simple NN with these initial weights gives very different optimized weights after every iteration. The optimization process is clearly very unstable.

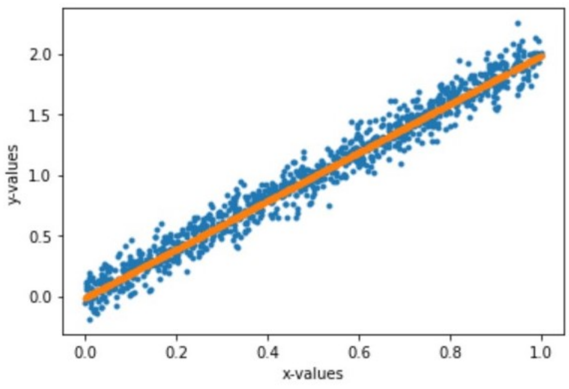
4.3 Use contour plots to visualize stability

In this section, we examine the stability of the Simple NN using contour plots. We remind ourselves that the Simple NN is equivalent to the interpolating function $N(x) = \alpha ReLU(\beta x + b) + h$. In the contour plot we let two weights vary inside some intervals and plot the respective loss of a combination of values for these two weights. We call a solution for the weights in the Simple NN stable, if it lies well within an area of values with low loss.
We start by fixing the values $b = h = 0$. To interpolate the data from the function $f(x) = \alpha^* x$ (with $\alpha^* = 2$), we know from the constraints in Section 3.2 that the only condition left for the neural network to fit $f(x)$ perfectly is that $\alpha \cdot \beta = 2$. Therefore we will plot each of these weights on the interval $[0, 4]$.
To code the contour plot, we first need to generate the data which should be plotted. More precisely, we need to calculate the losses for points in a grid of values inside the set $[0, 4]^2$, because this is the space in which we want to examine the weights α and β. Note that in this section we rely on Program 7 [2].
As visible in Figure 4.7, we use the function

```
Values_1, Values_2 = numpy.array(numpy.meshgrid(array_1,array_2))
```

to create a grid of points from two arrays `array_1` and `array_2`. The variable `Values_1` consists of all beta-values and `Values_2` consists of all alpha-values of the points in our grid. The function `numpy.array(...)` is necessary, to give the output from the `numpy.meshgrid(...)` function an appropriate form for how we will handle it in the rest of the program.

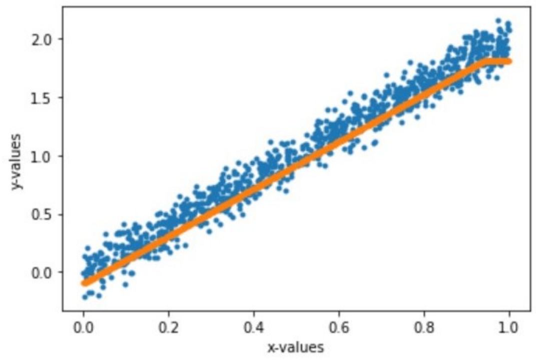
```
Epoch 50/50
50/50 [==============================] - 0s 1ms/step - loss: 0.0102
```



```
The Simple NN was initiliazed with the weights: alpha_init =  4  beta_init =  4  b_init =  -3  h_init =  -7
Constraints:
The horizontal part of the ReLU is left of the interval [0,1].
(i)  b/beta =  1.038321  =  2.2959123 / 2.2111776  >=? 0
(ii)  alpha_prime =  2  =?  1.999609  =  0.9043186 * 2.2111776  = alpha * beta
(iii)  h =  -2.0885944  =?  -2.076236  = - 0.9043186 * 2.2959123  = -alpha * b
```

(a) Results of initializing the Simple NN with a slightly perturbed solution. Notice that the Simple NN still optimizes well and the optimized weights are stable.

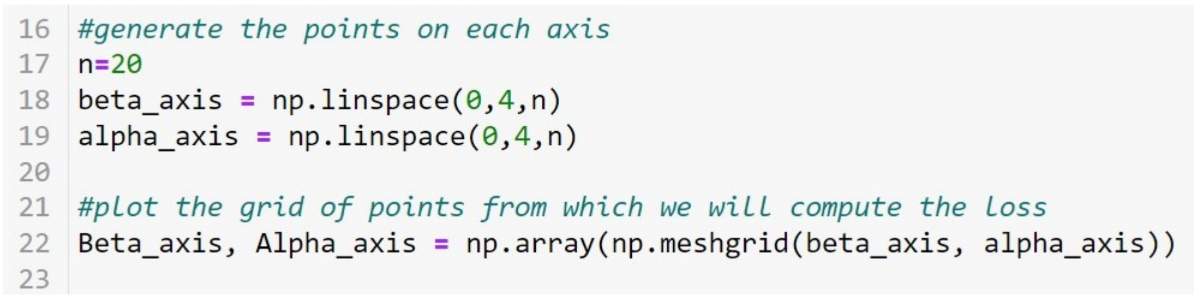
```
Epoch 50/50
50/50 [==============================] - 0s 977us/step - loss: 0.0269
```



```
The Simple NN was initiliazed with the weights: alpha_init =  -10  beta_init =  -1.5  b_init =  1.5  h_init =  1.5
Constraints:
The horizontal part of the ReLU is right of the interval [0,1].
(i)  b/beta =  -0.94213897  =  0.19705695 / -0.2091591  <=? -1
(ii)  alpha_prime =  2  =?  2.0272775  =  -9.692513 * -0.2091591  = alpha * beta
(iii)  h =  1.8120152  =?  1.9099771  = - -9.692513 * 0.19705695  = -alpha * b
```

(b) Results of initializing the Simple NN with a more perturbed solution. As the loss is closer to 0.02 than the minimal loss 0.01, the Simple NN optimized somewhat well. During the optimization we required more than 50 epochs to receive a good result. Further notice that optimized weights are very unstable.

Figure 4.6: Initializing the Simple NN with perturbed solutions for the weights and assessing how stable the optimized weights are. If we perturbate the initial values much more than in (b), the optimization fails frequently.

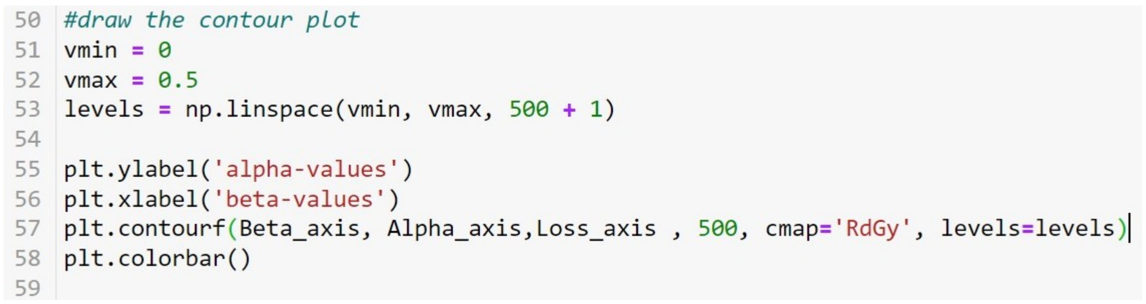


```python
#generate the points on each axis
n=20
beta_axis = np.linspace(0,4,n)
alpha_axis = np.linspace(0,4,n)

#plot the grid of points from which we will compute the loss
Beta_axis, Alpha_axis = np.array(np.meshgrid(beta_axis, alpha_axis))
```

Figure 4.7: Program 7 [2] - Creating the grid of points for each of which we will later compute the loss such that we can draw the contour plot.

Next, we iterate through all points in our grid, compute and save the respective losses. When we have generated the data, we are ready to plot it. The code for programming the contour plot is explained in Figure 4.8.



```python
50 #draw the contour plot
51 vmin = 0
52 vmax = 0.5
53 levels = np.linspace(vmin, vmax, 500 + 1)
54
55 plt.ylabel('alpha-values')
56 plt.xlabel('beta-values')
57 plt.contourf(Beta_axis, Alpha_axis,Loss_axis , 500, cmap='RdGy', levels=levels)
58 plt.colorbar()
59
```

Figure 4.8: Program 7 [2] - Drawing the contour plot. Note that beforehand we have imported the Python library `matplotlib.pyplot`. In lines 51 and 52 we set the maximal and minimal values which our contour plot should regard. We know that the loss cannot take on values below zero, and losses at ~ 0.35 are already considered baldy optimized. Thus `vmin` and `vmax` are chosen accordingly. `levels` decides how many steps our contour plot should have.

The final contour plot is depicted in Figure 4.9. Notice that the area of minimal losses very closely imitates the shape of the graph $L(\beta) = \frac{\alpha^*}{\beta} = \frac{2}{\beta}$. All points which lie on this graph perfectly fulfill the constraint $\alpha \cdot \beta = 2$. Thus it is plausible to see this result. Additionally the area of minimal loss is very slim. Any deviation of ~ 0.2 in the value of α or β would result in leaving the dark red area of minimal losses. But notice that the neural network is most stable at weights close to the point $(\sqrt{2}, \sqrt{2})$. We will further discuss this graph and the optimization of the weights α, β in Chapter 6.

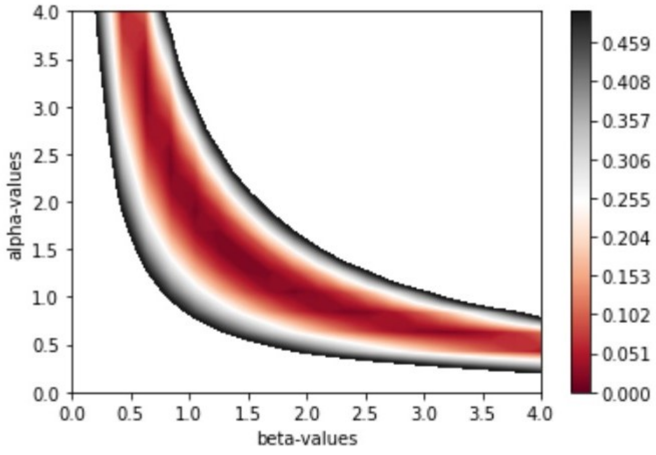


Figure 4.9: We plot the loss (MSE) of the Simple NN for points $(\beta, \alpha) \in [0, 4]^2$. We programmed the plot such that losses above the value of 0.5 are not considered. The white area in the plot corresponds to such losses. Note that we know that the loss is minimized when close to the value 0.01. According to the color bar, such values are attained for (β, α) inside the dark red area.

Chapter 5

Neural network stability of one hidden layer

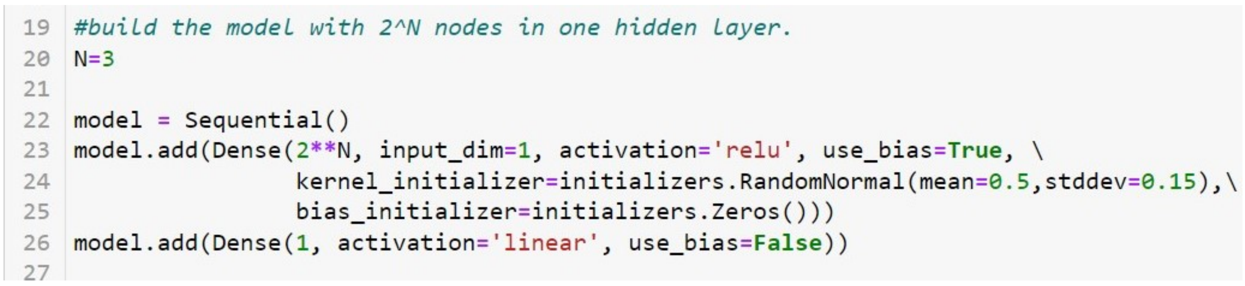
In this chapter we analyze neural networks, which are slightly more complex than the Simple NN. In particular, the neural network will have exactly one hidden layer with 2^N hidden nodes (where N can be chosen arbitrarily), with ReLU activation functions. Furthermore we add a bias node in the input layer. Essentially it is of the same form as the neural network in Figure 2.7. According to previous results, this neural network is the most effective of all architectures which we have seen so far in this report.
In the following sections we let this neural network interpolate the data from the functions x^2 and $\sin(2\pi x)$ on the interval $[0, 1]$ and analyze the stability of the optimized neural network, by drawing a contour plot, like in Section 4.3.

5.1 Interpolate the function x^2

In this section we discuss the code of Program 8 [2], which builds and fits the neural network, as explained above and then does an analysis of its stability. Several steps are similar to the analysis of the Simple NN in the previous chapters. The code is separated into two segments.
In the first segment we generate the data and fit the neural network. We start by generating the data of the function which we want to interpolate. This means, that like in Section 2.1, we start by generating 1000 random uniformly distributed points inside the interval $[0, 1]$. Then we generate the y-values in accordance with the function $h(x) = x^2$. Note that we will not add noise to the function. We continue by building the neural network, which we can see in Figure 5.1. A problem while programming the neural network was that repeatedly only about one third of the ReLU nodes were activated in the optimized neural network. The reason for this is that many nodes were initialized with weights which left the node inactive. Clearly the optimization was inefficient. But this could be resolved by including the attributes

```
kernel_initializer=initializers.RandomNormal(mean=0.5,stddev=0.15)
bias_initializer=initializers.Zeros()))
```

when adding the new layer. This ensures that the initial weights are randomly chosen with normal distribution closely around 0.5 and the bias weights are initialized with 0. Thus, during the initialization each ReLU function is activated and therefore not neglected during the optimization. To understand why the ReLU function is activated for such initial values, recall that our neural network is equivalent to the interpolation function $N(x) = \sum_{i=0}^{2^N-1} \alpha_i ReLU(\beta_i x + b_i)$. If the initial value β_i is positive and the initial value of b_i is zero (since b_i is the bias weight), then the term inside the ReLU function is positive for all $x \in (0,1]$. Thus each ReLU will be activated. In the program this yielded much better results after the optimization.



```
#build the model with 2^N nodes in one hidden layer.
N=3

model = Sequential()
model.add(Dense(2**N, input_dim=1, activation='relu', use_bias=True, \
                kernel_initializer=initializers.RandomNormal(mean=0.5,stddev=0.15),\
                bias_initializer=initializers.Zeros()))
model.add(Dense(1, activation='linear', use_bias=False))

```

Figure 5.1: Program 8 [2] - Build the neural network for $2^N = 2^3$ hidden nodes. Additionally to building previous neural networks, here we add a command about how the random initial values should be chosen during the optimization.

The neural network managed to fit the data well. In Figure 5.2 we plot the data and the performance of the neural network on the data.

```
Epoch 100/100
50/50 [==============================] - 0s 957us/step - loss: 2.4300e-04
```

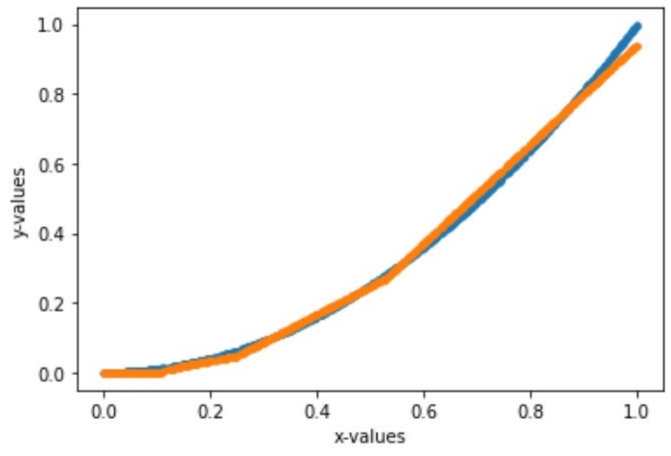
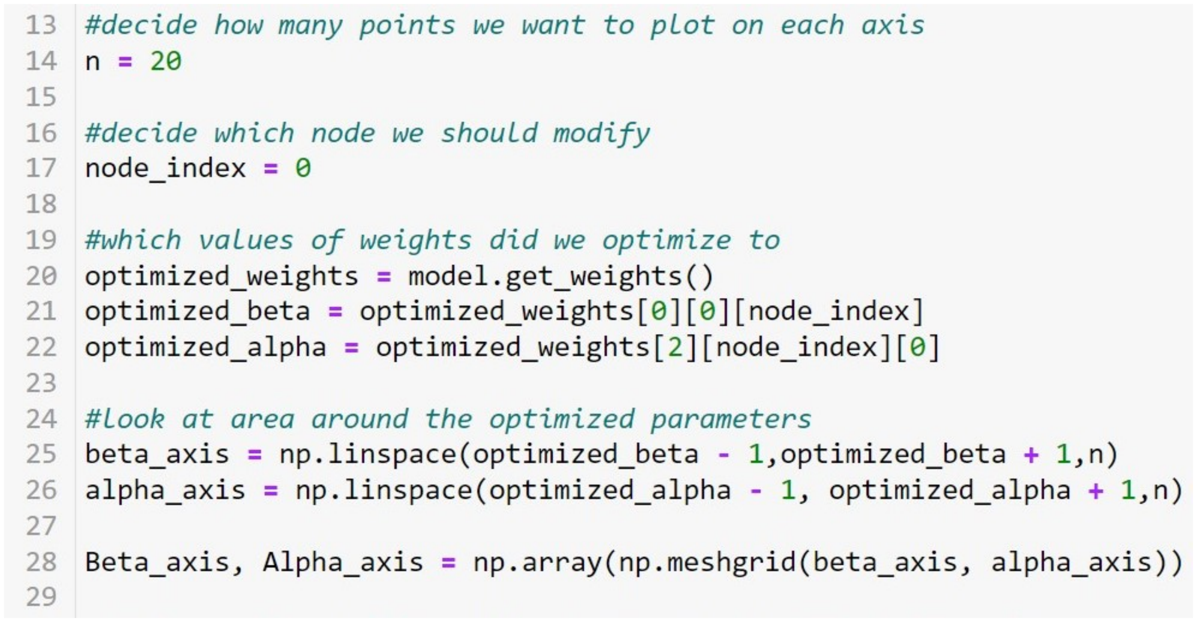


Figure 5.2: neural network with 2^3 hidden nodes (orange graph) in one hidden layer fits the function $h(x) = x^2$ on the interval $[0,1]$.

In the second segment of the program we analyze the stability of the neural network which we optimized. To achieve this, we draw the contour plot for the values α_i and

β_i of some node in the neural network i.e. for some $i \in \{0, ..., 2^N - 1\}$. After choosing the i-th node, we extract the weights which correspond to this node and generate a grid of points which form a square of side length 2 and the optimized weights (α_i, β_i) at the center. The code for generating this grid can be found in Figure 5.3. We will use this grid to compute the loss for points (α, β) which lie close to (α_i, β_i).



```python
#decide how many points we want to plot on each axis
n = 20

#decide which node we should modify
node_index = 0

#which values of weights did we optimize to
optimized_weights = model.get_weights()
optimized_beta = optimized_weights[0][0][node_index]
optimized_alpha = optimized_weights[2][node_index][0]

#look at area around the optimized parameters
beta_axis = np.linspace(optimized_beta - 1,optimized_beta + 1,n)
alpha_axis = np.linspace(optimized_alpha - 1, optimized_alpha + 1,n)

Beta_axis, Alpha_axis = np.array(np.meshgrid(beta_axis, alpha_axis))

```

Figure 5.3: Program 8 [2] - Creating the grid of points around the optimized weights α_i and β_i corresponding to the i-th node. In the program we choose i = `node_index = 0`. Then we regard values for β and α which have a distance of at most 1 to α_i and β_i respectively. In the last step we generate the meshgrid, like in previous codes for drawing a contour plot.

Afterwards, we draw the contour plot just like before and receive the plot in Figure 5.4. This shows us that the optimized neural network is somewhat stable. Clearly, changing the values of α_i and β_i by even a small amount of about ~ 0.1 in the wrong direction can already increase the loss significantly.

5.2 Interpolate the function $\sin(2\pi x)$

We can easily modify the program above to do the same fitting and analysis for the function $g(x) = \sin(2\pi x)$. Doing this, we receive Program 9 [2]. Fitting the neural network to $g(x)$ and plotting the predictions of the data as well as the original data, is illustrated in Figure 5.5. In Figure 5.6 we observe that the surface of weights with minimal loss is also somewhat stable. Since our optimized weights α_i and β_i lie exactly in the middle of the contour plot, thus well within the area shaded in red, which corresponds to a very low loss (MSE). Note that the optimized network reached a loss

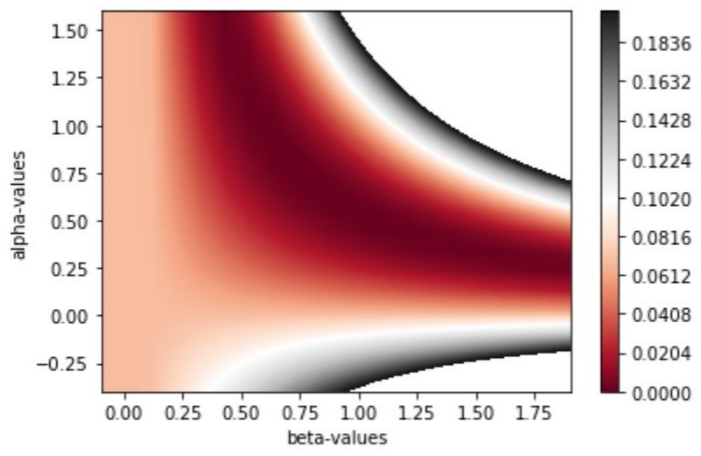
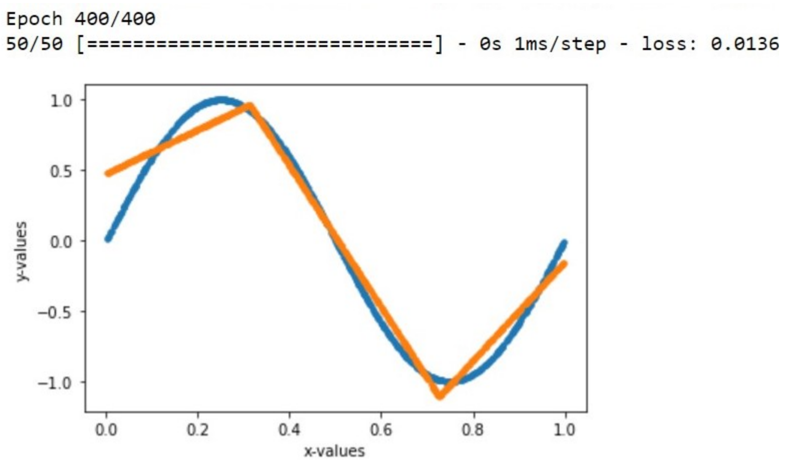


Figure 5.4: Contour plot of the neural network plotting the weights of the 0-th node against the loss (MSE).

of ~ 0.01 (see Figure 5.5), which is why we are mostly concerned with the dark red areas in the contour plot in Figure 5.6. But there do not appear to be large areas in dark red shade, which leads to the conclusion, that this optimized neural network is not very stable.



```
Epoch 400/400
50/50 [==============================] - 0s 1ms/step - loss: 0.0136
```

Figure 5.5: neural network with 2^4 nodes in one hidden layer fitting the function $g(x) = \sin(2\pi x)$ on the interval $[0, 1]$.

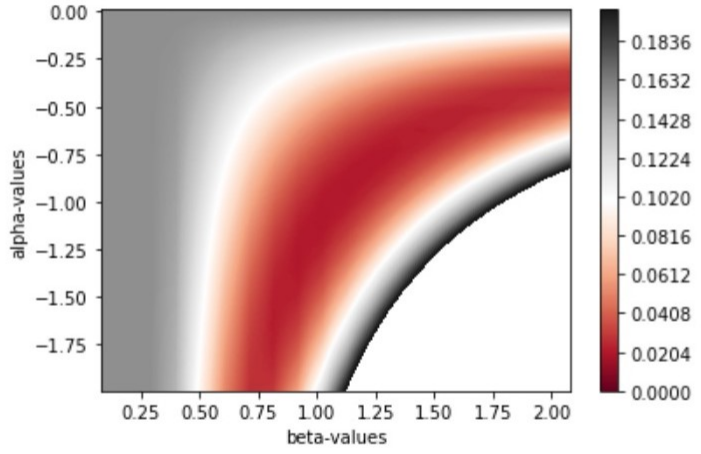


Figure 5.6: Contour plot of the 0-th node in the neural network fitting the function $g(x) = \sin(2\pi x)$.

Chapter 6

Behavior of optimization when varying the learning rate

In this chapter, we return to the examination of the Simple NN and fitting it to the linear function $f(x) = \alpha^* x$ (for $\alpha^* = 2$) with additive noise on the interval $[0, 1]$. In particular we will perform more detailed analyses of the stability of the neural network while modifying the learning rate.

6.1 Plot the optimized values for some initial weights

We remind ourselves that the Simple NN is equivalent to the interpolating function $N(x) = \alpha ReLU(\beta x + b) + h$. In this section, like before, we will fix $b = h = 0$, because then we land in Case 1 of the analytical solution of fitting $f(x)$ (see Section 3.2; note that $\alpha, \beta > 0$ has to be true as well). In fact, this way all constraints, except $\alpha \cdot \beta = \alpha^* = 2$ are automatically fulfilled. In this section, we analyze where some initial weights (β, α) are optimized to after the fitting process, when using different learning rates.
To do this, we draw a coordinate system with β-values on the x-axis and α-values on the y-axis (similarly to when drawing the contour plot). Since we know that $\alpha \cdot \beta = 2$ has to be fulfilled, to fit $f(x)$ perfectly, we know the loss of the Simple NN is minimal, if a point (β, α) lies on the graph $L(\beta) = \frac{\alpha^*}{\beta} = \frac{2}{\beta}$. Thus we can say that two initial weights (β, α) are well optimized, if their values after the fitting lie on the graph of $L(\beta)$. To illustrate this, we plot the graph of $L(\beta)$ and some points (β, α). Furthermore we will add some gradient lines. For this, note, that each line on which $\alpha \cdot \beta = const.$ is considered a level curve, since the loss is constant for all points on this line. $L(\beta)$ is such a level curve. Further note that gradients (thus also the gradient lines) are always perpendicular to level curves. This means, that we will plot lines perpendicular to the graph of $L(\beta) = \frac{2}{\beta}$ and which go through these points (β, α). An illustration of this can be found in Figure 6.1. Note that, in this section, we use Program 10 [2]. Since SGD iteratively updates its weights along the gradient, the optimized values of the plotted points should optimize close to where the gradient line intersects $L(\beta)$.

Before we continue, we want to choose the initial weights in a systematic way. To have more consistency, we would like the initial points to all have the same distance from

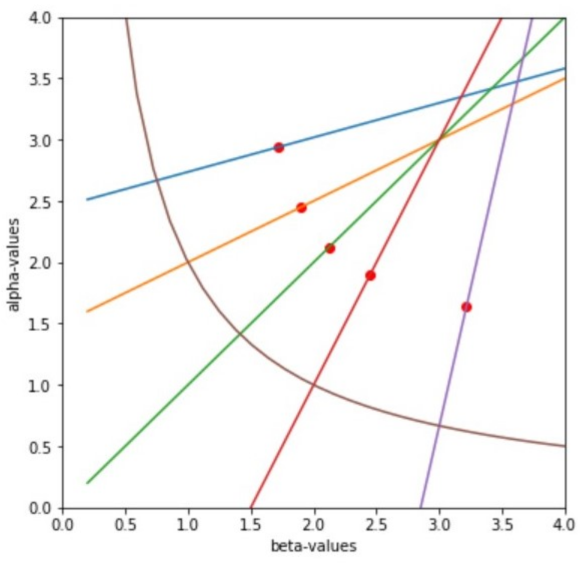


Figure 6.1: $L(\beta) = \frac{2}{\beta}$ is represented by the brown line. The red dots are some initial values of the weights and the straight lines are their respective gradient lines.

the graph. For this we take only an x-value and look at the point $(x, \frac{2}{x})$ (so the point which lies on the optimal fit-graph $L(\beta)$). Then as our initial values (β, α) we take the point which is exactly a distance d away from $(x, \frac{\alpha^*}{x})$, along its gradient line. A description of this can be found in Figure 6.2.

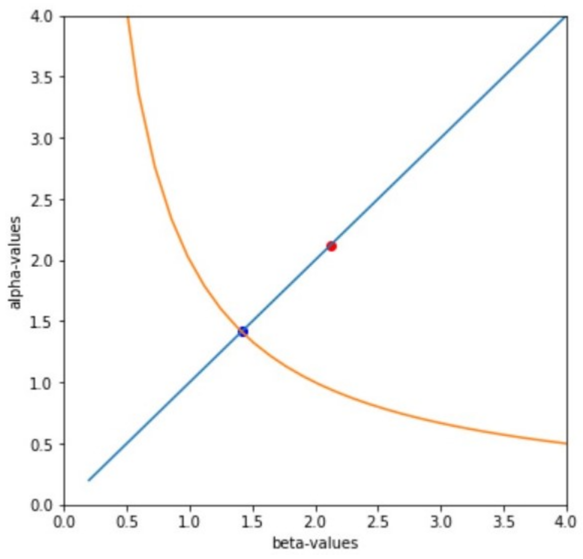


Figure 6.2: The blue point is the point which we receive when choosing some x-values. In dependence of this x-value, we want to find the coordinates of the red point, which is a distance of exactly d away from the blue point. Note that the blue line is the gradient line to the blue point i.e. perpendicular to $L(\beta)$.

We will now provide the derivation of the term of the gradient line at some base point $p = (x, \frac{\alpha^*}{x})$ on the graph of $L(\beta)$ (or $L(x)$). Note that we will call the blue point from Figure 6.2, $p = (p_x, p_y)$ and the red point $p' = (p'_x, p'_y)$. We let p_x be some arbitrary positive value, then since p lies on the graph of $L(x)$ it must hold that $p_y = \frac{\alpha^*}{p_x}$. Note that we want to find the term of a linear function $l_{p_x}(x) = ax + b$ which goes through the point p and has a perpendicular slope to $L(x)$ at the position p_x (this is the gradient line). This slope is given by $L'(p_x) = -\frac{\alpha^*}{p_x^2}$. For the slope of $l_{p_x}(x)$ to be perpendicular it must hold that

$$a \cdot L'(p_x) = -1 \Rightarrow a = \frac{p_x^2}{\alpha^*}.$$

Since p must lie on the graph of $l_{p_x}(x)$, we have that

$$l_{p_x}(p_x) = p_y \Rightarrow \frac{p_x^2}{\alpha^*} \cdot p_x + b = \frac{\alpha^*}{p_x} \Rightarrow b = \frac{\alpha^*}{p_x} - \frac{p_x^3}{\alpha^*}.$$

Thus the gradient line to the base point p is given by

$$l_{p_x}(x) = \frac{p_x^2}{\alpha^*}x + \frac{\alpha^*}{p_x} - \frac{p_x^3}{\alpha^*}.$$

Now let us derive the coordinates of the point p', which lies a distance of $d \geq 0$ away from p on the gradient line $l_{p_x}(x)$. First note that since p' lies on the graph of $l_{p_x}(x)$, it suffices to find p'_x , because we have that $p'_y = l_{p_x}(p'_x)$. So let us find p'_x , such that

$$\begin{aligned}
&d = \sqrt{(p'_x - p_x)^2 + (p'_y - p_y)^2} \\
&\Rightarrow d = \sqrt{(p'_x - p_x)^2 + \left(l_{p_x}(p'_x) - \frac{\alpha^*}{p_x}\right)^2} \\
&\Rightarrow d = \sqrt{(p'_x - p_x)^2 + \left(\frac{p_x^2}{\alpha^*} \cdot p'_x + \frac{\alpha^*}{p_x} - \frac{p_x^3}{\alpha^*} - \frac{\alpha^*}{p_x}\right)^2} \\
&\Rightarrow d^2 = (p'_x - p_x)^2 + \frac{p_x^4}{\alpha^{*2}}(p'_x - p_x)^2 \\
&\Rightarrow d^2 = \left(1 + \frac{p_x^4}{\alpha^{*2}}\right) \cdot (p'_x - p_x)^2 \\
&\Rightarrow p'_x - p_x = \pm\sqrt{\frac{d^2}{\left(1 + \frac{p_x^4}{\alpha^{*2}}\right)}} \\
&\Rightarrow p'_x = p_x \pm \sqrt{\frac{d^2}{\left(1 + \frac{p_x^4}{\alpha^{*2}}\right)}}.
\end{aligned}$$

Since, like in Figure 6.2, the red point p' should be to the right of the blue point p, we want p'_x to be larger than p_x. Thus we can say that p' must be given by

$$p'_x = p_x + \sqrt{\frac{d^2}{\left(1 + \frac{p_x^4}{\alpha^{*2}}\right)}}$$

Having these formulas makes it much more convenient to generate many points at the same time and plot where they optimize to, by simply choosing several different x-values. These points are also to some degree comparable, since they keep the same distance to the graph (along their respective gradient line). We implement these formulas into a program and then plot all relevant information which we have talked about before. The result of some examples of points can be found in Figure 6.3

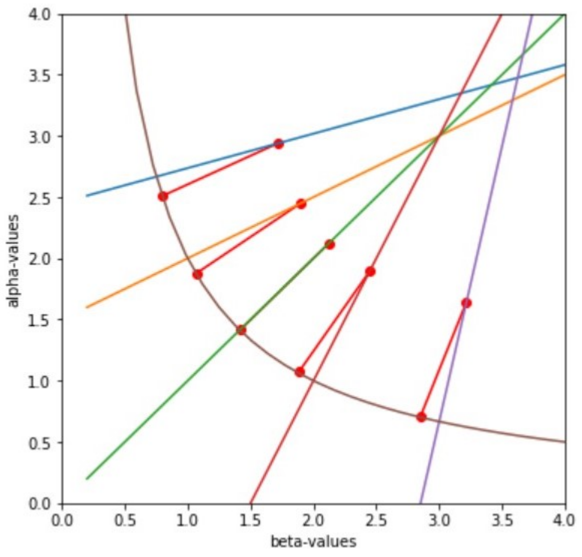
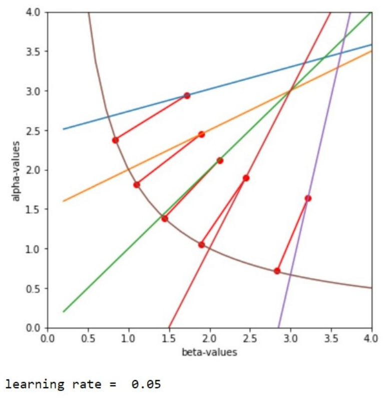
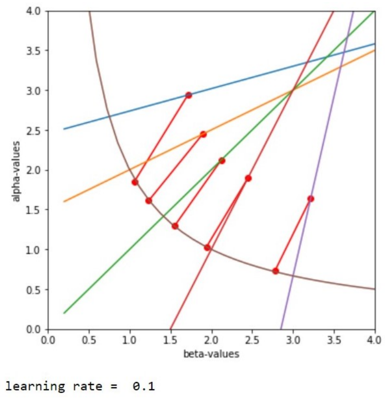


Figure 6.3: Every initial point is connected by a line to its optimized weight in the Simple NN. In this case we compiled each optimization with a learning rate of $\eta = 0.01$. Note hate we compiled the program for the the points with the x-values $0.75, 1.0, 1.41421356237, 2, 3$ (these are the base points p respectively). These points are evenly spaced along the graph of $L(x)$ and we include an x-value, which is close to $\sqrt{2}$, because the most stable solution of the weights is given by the point $(\sqrt{2}, \sqrt{2})$. Its gradient is given by the green line.

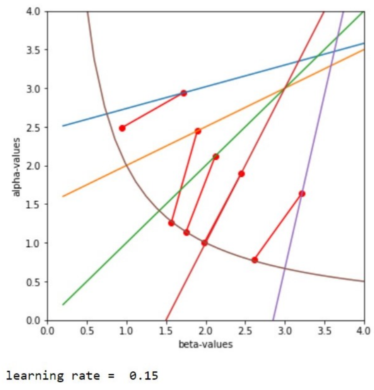
In the following we examine how this plot changes, as we vary the learning rate. An overview of the results is given by Figure 6.4. Here we notice that as the learning rate increases, the initial values seen to optimize towards some specific point on the graph of $L(x) = \frac{2}{x}$.



(a) Results of the optimizations with a learning rate of $\eta = 0.05$



(b) Results of the optimizations with a learning rate of $\eta = 0.1$



(c) Results of the optimizations with a learning rate of $\eta = 0.15$. While one point does not optimize well, the other points a clear tendency to some point on $L(x)$.

Figure 6.4: We apply different learning rates and plot where the weights are optimized to. As we can see, for different learning rates, the weights seem to optimize to different optimal weights along the graph of $L(x) = \frac{2}{x}$. Notice that for larger learning rates, there appears to be a trend in the direction which weights optimize towards.

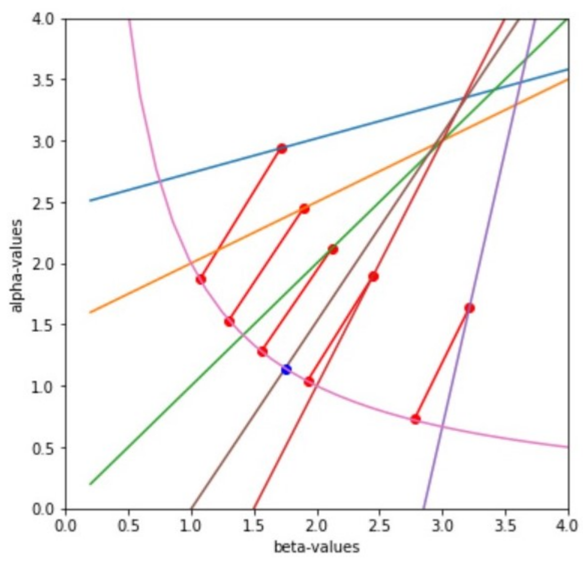
6.2 Unexpected tendency of optimized weights (α, β)

In this section we discuss the previous observation from Section 6.1. We noticed, that for small learning rates like 0.01 or 0.005, the initial weights tend to be optimized slightly towards the point $(\sqrt{\alpha^*}, \sqrt{\alpha^*}) = (\sqrt{2}, \sqrt{2})$ i.e. the most stable solution. Thus the optimization seems to stabilize the optimized solutions of the initial weights when a small learning rate is applied. On the other hand with a larger learning rate like $\eta = 0.15$ we see a clear tendency towards a point close to $(2, 1)$. Thus for larger learning rates, the instability of the optimized solution can increase, even if it fits the data well. This unexpected tendency leads us to believe that the learning algorithm prefers modifying the initialized weight α during optimization, rather than changing the value of the initial weight β. Further note that the constraint $\alpha \cdot \beta = \alpha^*$ is symmetric with respect to α and β. Thus, clearly the symmetry in the solution is broken. This might be a sign of possible impartiality of the learning algorithm within the weights of different layers during the fitting process.

When further analyzing the plots in Figure 6.4, it appears that for different learning rates, the point which the optimized weights seem to tend to, moves along the graph of $L(x)$. In particular in the graph in Figure 6.4 (b), the second point from the right optimizes slightly to the left of its gradient line. But in the graph (c) this point optimizes exactly onto its gradient line. This gives the impression that the tendency in both of these graphs is slightly different. To get a better sense of this tendency in a systematic way, we assume there exists such a tendency point T for every learning rate, which we will examine more closely. Note that for small learning rates, this tendency point T is clearly very close to $(\sqrt{2}, \sqrt{2})$. See figure 6.3.

After every optimization we identify the pattern that there exists a point T on the graph of $L(x)$, such that all weights that are initialized to the left of the gradient line of T, optimize to some point to the right of the gradient line of the initial weights. Thus we can say that they optimize "towards" the point T. The same thing happens analogously for the points which are initialized to the right of the gradient line of the point T. An illustration of such a tendency point can be found in Figure 6.5.

In the following we describe Program 11 [2], which, among other things, finds a good estimate for the x-value of this tendency point T. The program checks for every point of initial weights, if these weights optimized to the left or to the right of the gradient line on which their initial values lied. This information is stored in a Boolean vector with the name `bool_opt`. While iterating through all initial weights and fitting them, we set `bool_opt[i] = 0` if the i-th node optimizes to the left of its initial gradient line, `bool_opt[i] = 1` if the i-th node optimizes to the right of its initial gradient line, and `bool_opt[i] = 2` if the i-th node optimizes very closely to its initial gradient line.
If the previously described pattern of the existence of a tendency point is fulfilled, then the tendency point will lie between the two adjacent gradient lines for which the following holds true: the weights which were initialized on the left gradient line optimized to the right of this initial gradient line. And the weights which were initialized



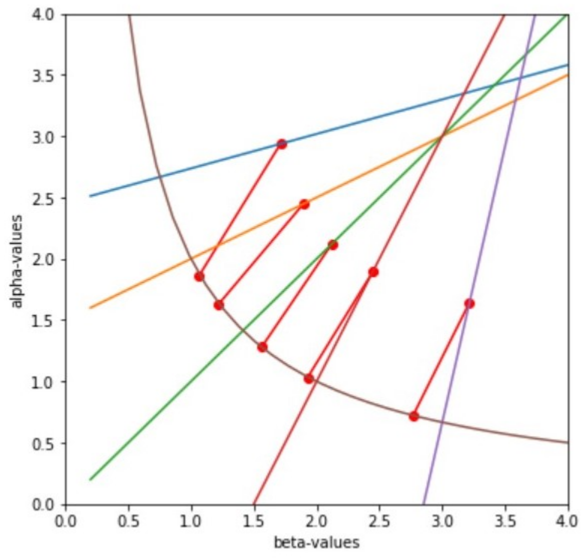
```
learning rate =  0.1
```

Figure 6.5: An illustration of a tendency point T is given by the blue point in the graph. All initial weights seem to optimize away from their own gradient line and towards T.

on the right gradient line optimized to the left of this initial gradient line. This is e.g. the case for the green and red gradient lines in Figure 6.5. In this example the Boolean vector would have the form `bool_opt = [1.  1.  1.  0.  0.]`.

In a more mathematical sense this means that we choose some points on the x-axis $x_0 < x_2 < ... < x_n$ and for these points we run the program, which is described in Section 6.1. Then we receive the pattern, that up to some x-value x_i (for $i \in \{0, ..., n\}$), all initial weights optimized to the right of their gradient line, and after x_i the initial weights optimize to the left of their respective gradient line. In this case, we can only know that the tendency point lies within $[x_i, x_{i+1}]$. We decide that as long as we have enough points (and for for n large enough and all x_i acceptably distributed on the interval) then the term $\frac{x_i + x_{i+1}}{2}$ is a good estimate for the x-value of the tendency point T. A result of such a program using the same examples which we used previously in this chapter can be found in Figure 6.6.

```
Boolean vector: bool_opt =  [1. 1. 1. 0. 0.]
The x-value of the tendency point T is approx.  1.707106781185
```



```
learning rate =  0.1
```

Figure 6.6: This shows us that the Boolean vector `bool_opt` is exactly of the expected form. The estimate for the x-value of the tendency point T is given exactly by $\frac{1.414213562337+2}{2} = 1.707106781185$.

6.3 Plot the learning rate against the x-value of the tendency point T

In this section we use previous tools to analyze how the tendency point T behaves, as we change the learning rate. In Section 6.2 we suspected that the tendency point moves along the graph of $L(x) = \frac{2}{x}$ (with increasing x-values), as we increase the learning rate. In the following, we will test this theory, using Program 12 [2], which iterates through a set of learning rates and for each learning rate computes an estimate for the x-value of the tendency point. The program then plots the learning rate on the x-axis and the x-value of the tendency point on the y-axis. Note that we are especially interested in what happens with smaller learning rates. In the program we had to pay attention to the case in which the optimization of the points did not go well for some learning rate. This is a problem, because when computing the learning rate we relied on the assumption that the weights optimized onto the graph of $L(x)$. If this is not the case, then the computation of the x-value of the tendency point will be distorted. Thus, in our program, we automatically set the x-value of the tendency point to be zero, if one point did not optimize close to $L(x)$. Therefore we note that if the x-value of the tendency point is zero for some learning rate, then this means that at least one point did not optimize well onto $L(x)$.
A result of this program using the same examples as in the previous sections is given in Figure 6.7.

We caution the reader judge these results carefully, since we analyzed only a small

```
x-values:  [0.75       1.         1.41421356 2.         3.        ]
learning rates:  [0.005 0.01  0.05  0.1   0.15 ]
```

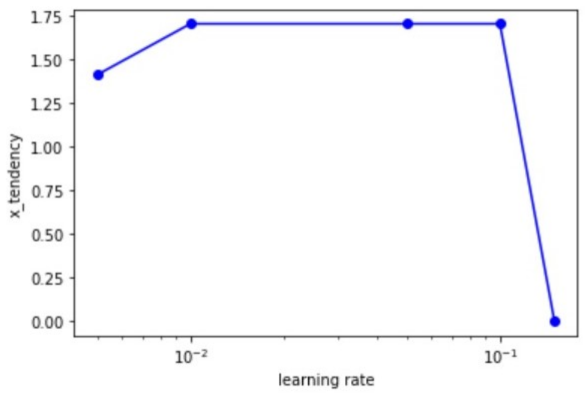
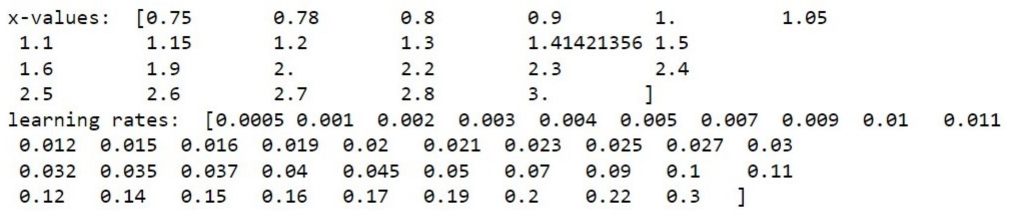


Figure 6.7: This is a plot of the same x-values which we have used in the previous sections. Note that for the smallest learning rate $\eta = 0.005$ the tendency point seems to be close to $\sqrt{2}$ and for all larger learning rates the x-value of the tendency point moves along the graph of $L(x)$ to a larger x-value.

number of points in our computation. To address this problem we compute the same graph for a larger number of x-values and learning rates. The result of this compilation can be found in Figure 6.8. In that plot we see that for small learning rates, the x-value of the tendency point is very close to $\sqrt{2}$. As we increase the learning rate we can observe that the tendency point T moves along the graph of $L(x)$ to larger x-values. While we do not consider this to be concluding evidence, this observation supports our previous theory.
In summary, it appears that for larger learning rates, we observe that the optimization of the weights can have a tendency towards a less stable setting of weights in the Simple NN. For smaller learning rates (between 0.001 and 0.35), the optimization shows a tendency towards more stable weights.



```
x-values:  [0.75       0.78       0.8        0.9        1.         1.05
 1.1        1.15       1.2        1.3        1.41421356 1.5
 1.6        1.9        2.         2.2        2.3        2.4
 2.5        2.6        2.7        2.8        3.        ]
learning rates:  [0.0005 0.001  0.002  0.003  0.004  0.005  0.007  0.009  0.01   0.011
 0.012  0.015  0.016  0.019  0.02   0.021  0.023  0.025  0.027  0.03
 0.032  0.035  0.037  0.04   0.045  0.05   0.07   0.09   0.1    0.11
 0.12   0.14   0.15   0.16   0.17   0.19   0.2    0.22   0.3   ]
```

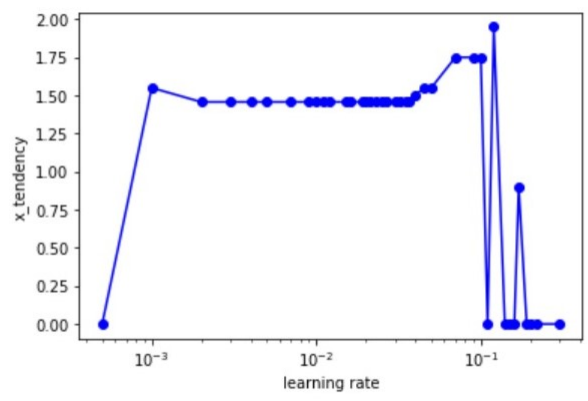


Figure 6.8: This is a plot of several learning rates against the x-values of the respective tendency points which the program approximated.

Chapter 7

Conclusion

This report analyzes how well different architectures of neural networks optimize to interpolate a set of data points. We realize that it is preferable to structure many ReLU activated nodes in one layer, rather than spreading them out over several layers. Subsequently, we analyze the ReLU function in more detail and use the gained insight, to construct some constraints which give us information on how well and stable a neural network was fit to some data.
After this, we examine the stability of the Simple NN and confirm our expectation that initializing the weights with unstable or perturbed solutions of the constraints leads to a poor capability of the network to optimize, or to highly unstable solutions. To further assess the stability of neural networks, we draw contour plots, which visualize the performance of different network architectures trying to fit some data.
Finally we examine where weights of the Simple NN are sent after optimization with different learning rates. We do this, by plotting the initial weights and their optimized values in a coordinate system. We notice that all optimized points seem to tend towards a specific tendency point, which breaks the symmetry of the solution. This can be an indicator of an impartiality of the learning algorithm. Further, we find that the tendency point seems to move as we increase the learning rate. We quantify this, by computing an estimate of this tendency point for each learning rate and then plotting the respective learning rate against the x-value of the tendency point. This plot supports our previous observation.

Bibliography

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

[2] P. Lahmann, "Stability-of-Simple-Neural-Networks," 12 2021. [Online]. Available: https://github.com/PabloLah/Stability-of-Simple-Neural-Networks

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Stability of Simple Neural Network Architectures

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Lahmann	Pablo

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
30.12.2021	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.