Analysis of neural network performances in regression and classification problems
FYS-STK4155 - Project 2

Lars Bosch, Philipp Brückelt, Thomas Engl*
University of Oslo
(Dated: November 10, 2025)

We construct a neural network which can be used for the approximation of d-dimensional functions and image recognition. For the former, we test our network on two different examples, namely the Runge and Rastrigin function. Our results show that with a suitable choice of parameters and using mainly sigmoid as activation function, the neural network achieves an almost optimal mean squared error, equal to the variance of the noise of the data. It thus can compete with analytic or other numerical methods as ordinary least squares or Ridge and LASSO regression. Moreover, we find that for simple regression tasks, a small number of hidden layers and nodes per layer is sufficient. In addition, neural networks reduce oscillations of the solution at the boundary and can also approximate high-dimensional functions. For classification, we analyze the accuracy of the network on the MNIST data set. We obtain multiple good models by using ReLU in the first layer and sigmoid in the following and halving the number of hidden nodes in every layer. The best model in our comparison has hidden layers with 512 and 256 with a test accuracy of 97.8 % which is the most complex model that we discussed.

I. INTRODUCTION

The development of neural networks has led to a significant improvement in the numerical solving of mathematical problems or classification tasks in recent years. They are an integral part of machine learning and artificial intelligence and allow for solving problems for which no algorithms existed before, as well as yield new and more efficient approaches for ones where solutions exist.

A neural network [1], also called an artificial neural network, is a machine learning model inspired by biological neural network. The main idea is that input data is processed by affine-linear transformations and non-linear so-called activation functions. The data propagates through the network and passes different layers where the nodes of consecutive ones are connected by edges that model the synapses of a brain. The layers between the input and output layer are called hidden layers and if there are at least two of those, we usually refer to a deep neural network. Figure 1 shows an example of a neural network with two hidden layers. Each of them admits four nodes, the input size is 3, the output size 2.

The output of the network shall predict the correct target data. This is achieved by training the network where the network learns the connections between training data and corresponding targets. Afterwards it should be able to translate the learned rules to new data. Technically speaking, in the training, the various parameters of the network are determined such that the difference between output and targets of the train data is minimized. The latter is measured by a cost function and the minimization is done using numerical optimization methods [2] such as gradient descent or variants of it.

The parameters of a neural network are particularly the weights and biases with which the nodes of the layers are computed. The number of those depends on the the number of hidden layers and nodes per layer. Furthermore, the choice of activation functions drastically influences the accuracy of the predictions. Also, we can introduce another hyper parameter, by using regularization in the optimization problem.

A big research area in machine learning is the fitting of data by suitable continuous functions. A famous approach to do this is, e.g., linear regression, where one fits given data by a polynomial of a certain degree. However, this clearly restricts the range of solutions to a small subspace of continuous functions. In contrast, a neural network can find a curve fitting best to the given data that does not necessarily describe a polynomial and, hence, admits a larger range of approximating functions. In fact, by the universal approximation theorem [3, 4], any continuous function can be arbitrarily well approximated by a neural network with a single hidden layer with sufficiently many nodes, provided that the activation function is non-polynomial.

In this project, we want to construct a neural network that can approximate an arbitrary d-dimensional continuous function, where $d \in \mathbb{N}$. We start with simple one-dimensional functions. In particular, we study the Runge function [5]. Afterwards, we extend our numerical experiments to more sophisticated examples. The goal of this work is to find the optimal parameters of the network, to minimize the difference between the original data and our predictions on the test data.

Afterwards, we turn to classification tasks. In fact, we want to analyze the performance of a neural network at handwritten recognition, a classification problem which has been studied for around 100 years. Here we use the MNIST data set [6] which consists of 70,000 handwritten numbers from 0 until 9, each an image with 28 x 28 pixels like in Figure 2. For example [7] achieve with Convolutional Neural Networks a test accuracy of over 99.8%. The accuracy simply counts how many samples

* https://github.com/thomas-engl/Project2-Applied-ML

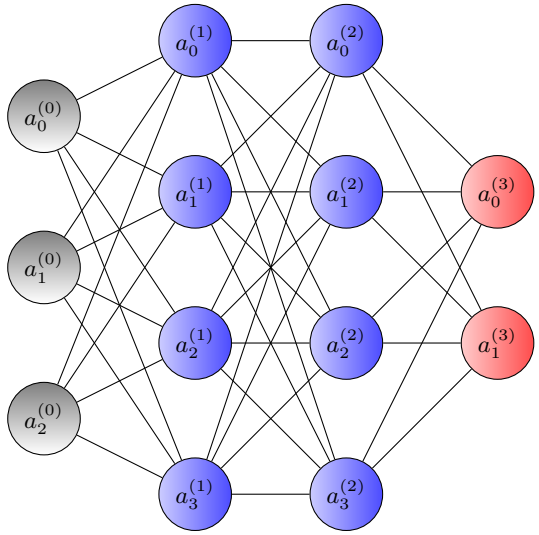

Figure 1. Architecture of a neural network with input size 3, two hidden layers, both with 4 nodes, and output size 2.

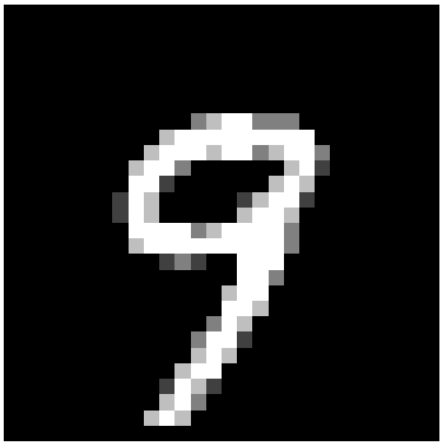

Figure 2. Sample from the MNIST Dataset: True value 9

are classified correctly which can be formalized as

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n}, \tag{1}$$

where I is the indicator function. We want to discuss how a simple feed forward neural network can handle this dataset. Therefore, we explain the pre-processing of the data, how to find suitable hyper parameters for this neural network and evaluate the performance of the final neural network. Interesting insights will be given by the confusion matrix which shows which numbers are confused with others.

The remainder of this report is structured as follows. In Section II we provide a deeper mathematical background of neural networks and their parameters. Thereafter, Section III deals with the implementation of the neural network in python. In Section IV we present the results of our numerical experiments. Finally, we summarize the main findings in Section V.

II. MATHEMATICAL BACKGROUND

A. Architecture of a neural network

We have already roughly described the architecture of a neural network in the Introduction. Now, we want to dive deeper into the mathematics. A neural network is fed by input data $\mathbf{x} \in \mathbb{R}^d$, say, for some $d \in \mathbb{N}$. We refer to d as the input size and $n \in \mathbb{N}$ as the number of inputs. Within the network, the input is transformed by linear operations and a non-linear activation function σ. The latter takes as input the output of each neuron. It allows the neural network to also fit non-linear functions and can help normalizing the values from each neuron.

Given the input $\mathbf{x}$, a node in the first hidden layer is computed as

$$z_j^{(1)} = \sum_{i=1}^{d} w_{ij} x_i + b_j, \quad j = 1, ..., m$$

where $\mathbf{W} = (w_{ij})_{i,j=1}^{d,m} \in \mathbb{R}^{d \times m}$ and $\mathbf{b} = (b_j)_{j=1}^{m} \in \mathbb{R}^m$ are the weights and the bias, respectively. Here, m is the size of the current layer. w_{ij} denotes the weight of the connection between x_i and x_j. In short, $\mathbf{z} = (z_j)_{j=1}^{m}$ can be written as

$$\mathbf{z} = \mathbf{W}^\mathsf{T}\mathbf{x} + \mathbf{b}. \tag{2}$$

Afterwards, the activation is applied component-wise, and we obtain

$$\mathbf{a} = \sigma(\mathbf{z}) \tag{3}$$

as the output of the layer. The activation functions that we use in our network are discussed in Subsection II C. Next, $\mathbf{a}$ is fed into the next layer and we proceed as before, with $\mathbf{a}$ instead of $\mathbf{x}$. The output of the last layer, the output layer is evaluated by the cost function C. Examples of those can be found in Subsection II D.

We first initialize the weights and bias with random numbers. The network is then trained using the training data. Thereby, we compute the derivatives of the cost function with respect to the weights and the biases using the chain rule. Afterwards, we search for the optimal parameters using gradient descent.

When working with neural networks, the number of parameters of the model grows rapidly. For example, if we have only two hidden layers with 50 and 100 nodes and input and output sizes both 1, e.g., for approximating one-dimensional functions, we have three weight matrices $\mathbf{W}^{(0)} \in \mathbb{R}^{1\times 50}, \mathbf{W}^{(1)} \in \mathbb{R}^{50\times 100}$ and $\mathbf{W}^{(2)} \in \mathbb{R}^{100\times 1}$ as well as three bias vectors $\mathbf{b}^{(0)} \in \mathbb{R}^{50}, \mathbf{b}^{(1)} \in \mathbb{R}^{100}$ and $\mathbf{b}^{(2)} \in \mathbb{R}$. This gives a total number of $5,301$ parameters to train. The number becomes clearly larger for deeper neural networks with more nodes per layer.

In our numerical experiments, we always work with batched inputs. That is, we do not only feed the network with one single input but with several at one time. For example, when we want to approximate a function using n data points in the batch, we process all data points at once. In case of a function on a d-dimensional domain, the input of the network is then a matrix $\mathbf{X} \in \mathbb{R}^{n\times d}$. Hence, equation 2 becomes

$$\mathbf{z} = \mathbf{X}\mathbf{W} + \mathbf{B}$$

where $\mathbf{B} \in \mathbb{R}^{n\times m}$ is a matrix with the vector $\mathbf{b}$ in every column.

B. Backpropagation

To train the network, we use gradient descent to find the optimal network parameters minimizing the cost function. To this end, we need the derivatives of the cost function w.r.t. the parameters, i.e., weights and biases. Computing these directly, is very hard, but we can make use of the chain rule for efficient calculations. This procedure is called back-propagation [1, Chapter 2].

We need two assumptions on the cost function s.t. back-propagation works: First, it is necessary that it can be written as average over cost functions for single training samples. Secondly, we need that the cost function, which is actually a function of the network parameters, can be written as a function of the network outputs. In Subsection II D it becomes clearer why these assumptions are important and one can see that the cost functions that are used in this work satisfy the assumptions.

Now, assume we have n layers, $n \in \mathbb{N}$. Let $\mathbf{W}_i$ and $\mathbf{b}_i$ be the weights and bias, respectively, of the i-th layer, $i = 1, ..., n$. By the chain rule, we have

$$\frac{\partial C}{\partial \mathbf{W}_i} = \frac{\partial C}{\partial \mathbf{a}_i}\frac{\partial \mathbf{a}_i}{\partial \mathbf{z}_i}\frac{\partial \mathbf{z}_i}{\partial \mathbf{W}_i}.$$

and

$$\frac{\partial C}{\partial \mathbf{b}_i} = \frac{\partial C}{\partial \mathbf{a}_i}\frac{\partial \mathbf{a}_i}{\partial \mathbf{z}_i}\frac{\partial \mathbf{z}_i}{\partial \mathbf{b}_i}.$$

If $i = n$, the first derivative $\frac{\partial C}{\partial \mathbf{a}_i}$ is simply the derivative of the cost function w.r.t. to the output, which can be easily computed, see Subsection II D. If $i \neq n$, we compute the derivative as

$$\frac{\partial C}{\partial \mathbf{a}_i} = \frac{\partial C}{\partial \mathbf{z}_{i+1}}\frac{\partial \mathbf{z}_{i+1}}{\partial \mathbf{a}_i}.$$

From equations (3) and (2), we find that

$$\frac{\partial \mathbf{a}_i}{\partial \mathbf{z}_i} = \sigma'(\mathbf{z}_i), \tag{4}$$

$$\frac{\partial \mathbf{z}_i}{\partial \mathbf{W}_i} = \mathbf{a}_i^\mathsf{T}, \tag{5}$$

$$\frac{\partial \mathbf{z}_i}{\partial \mathbf{b}_i} = \mathbf{1}, \tag{6}$$

$$\frac{\partial \mathbf{z}_{i+1}}{\partial \mathbf{a}_i} = \mathbf{W}_{i+1}^\mathsf{T}. \tag{7}$$

The equations are to be understood element-wise where it makes sense.

C. Activation functions

We will mainly work with the following activation functions:

(i) ReLU:

$$\sigma_1(z) := \begin{cases} z, & \text{if } z \geq 0, \\ 0, & \text{if } z < 0. \end{cases}$$

(ii) Leaky ReLU:

$$\sigma_2(z) := \begin{cases} z, & \text{if } z \geq 0, \\ \alpha z, & \text{if } z < 0, \end{cases}$$

where $\alpha > 0$.

(iii) Sigmoid:

$$\sigma_3(z) := \frac{1}{1+\mathrm{e}^{-z}}.$$

(iv) Softmax: In contrast to the activation functions before, the softmax function takes a vector as input. It is defined by

$$\sigma_4 \colon \mathbb{R}^d \to \mathbb{R}^d, \quad \sigma(\mathbf{z})_j := \frac{\mathrm{e}^{z_j}}{\sum_{k=1}^d \mathrm{e}^{z_k}},$$

for $j = 1, ..., d$.

For using the first three activation functions in our network, we also need their derivatives. Softmax is, however, only used in combination with the cross-entropy cost function, see Subsection II D. In this case, we do not need to compute the derivatives of the two functions separately. We discuss the details later.

For the ReLU function, the derivative is given by

$$\sigma_1'(z) = \begin{cases} 1, & \text{if } z \geq 0, \\ 0, & \text{if } z < 0 \end{cases}$$

and for Leaky ReLU we have

$$\sigma_2'(z) = \begin{cases} 1, & \text{if } z \geq 0, \\ \alpha, & \text{if } z < 0. \end{cases}$$

In both cases, the derivative does only exist in the distributional sense. The derivative of the sigmoid function is computed as

$$\begin{aligned} \sigma_3'(z) &= -(1+\mathrm{e}^{-z})^{-2} \cdot (-\mathrm{e}^{-z}) \\ &= \frac{\mathrm{e}^{-z}}{(1+\mathrm{e}^{-z})^2} \\ &= \frac{1}{1+\mathrm{e}^{-z}} - \frac{1}{(1+\mathrm{e}^{-z})^2} \\ &= \sigma_3(z)(1-\sigma_3(z)). \end{aligned} \tag{8}$$

The identity can also be used as an activation function. It is particularly useful in the output layer for regression problems.

D. Cost functions

In our experiments, we work with the following cost functions:

(i) Mean squared error. This cost function is used in our regression problems. Given the predictions $\tilde{\mathbf{y}}$ and the target data $\mathbf{y}$, it is given by

$$C_1(\mathbf{y}, \tilde{\mathbf{y}}) := \sum_{i=1}^{n} \frac{1}{n}(\tilde{y}_i - y_i)^2$$

(ii) Binary cross-entropy: We use this function in binary classification problems. Let $\tilde{y}$ be the predicted probability of the positive class y. Then, the binary cross-entropy function is given by

$$C_2(y, \tilde{y}) := -(y \log \tilde{y} + (1 - y) \log(1 - \tilde{y}))$$

(iii) Multiclass cross-entropy: It is a generalization of C_2 and is used for multiclass classification problems. Let y_i be the true label of the i-th class and $\tilde{y}_i$ the predicted probability, $i = 1, ..., n$ for some $n \in \mathbb{N}$. Then the multiclass cross-entropy is defined by

$$C_3(\mathbf{y}, \tilde{\mathbf{y}}) = -\sum_{i=1}^{n} y_i \log \tilde{y}_i.$$

For simplicity, we defined the cost functions as functions of the output $\mathbf{y}$ and targets $\tilde{\mathbf{y}}$, but, actually, the cost function of a neural network depends on all its parameters. We denote the parameters by θ in the following.

To train the network, we need the derivatives of the cost function with respect to the weights and biases. In practice, we only compute the derivative with respect to the output of the network and then obtain the derivative with respect to the weights using the chain rule in backpropagation, as done in Subsection II B.

The derivatives of the cost functions mentioned before w.r.t. the predictions are

(i) $\nabla C_1(\mathbf{y}, \tilde{\mathbf{y}}) = \frac{2}{n}(\tilde{\mathbf{y}} - \mathbf{y})$,

(ii) $\nabla C_2(y, \tilde{y}) = -\frac{y}{\tilde{y}} + \frac{1-y}{1-\tilde{y}}$,

(iii) $\nabla C_3(\mathbf{y}, \tilde{\mathbf{y}}) = -\frac{\mathbf{y}}{\tilde{\mathbf{y}}}$,

where the last equation is meant component-wise. In practice, we do not need the derivative of the cross-entropy cost function if the activation function of the output layer is softmax. Indeed, the product of the derivative of cross-entropy and softmax reduces to $\mathbf{y} - \tilde{\mathbf{y}}$ [8].

To avoid overfitting, we can perform a regularization. Then, the cost function is given by

$$C(\theta) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\theta) + \lambda \|\mathbf{W}\|_p^p$$

where $\mathcal{L}$ is the loss function, θ contains the parameters of the network and $p \in \{1, 2\}$. The approach resembles Ridge and LASSO regression. For $p = 2$, we obtain the Frobenius norm, i.e.,

$$\|\mathbf{W}\|_2^2 = \sum_{i,j} w_{ij}^2.$$

If $p = 1$, we sum over the absolute values of the entries, i.e.,

$$\|\mathbf{W}\|_1 = \sum_{i,j} |w_{ij}|.$$

The derivative of the cost function is then the sum of the derivative of the loss function and the one of the norm multiplied by the hyper parameter λ. Hence, if $p = 2$,

$$D_{\mathbf{W}} \|\mathbf{W}\|_2^2 = 2\lambda \mathbf{W}$$

and if $p = 1$, we have

$$D_{\mathbf{W}} \|\mathbf{W}\|_1 = \lambda \operatorname{sgn}(\mathbf{W})$$

where the sign function sgn is applied component-wise.

If we use regularization for our numerical experiments, we compute the gradient of the cost function as usual via back propagation and add the derivative of the regularization term at the end.

One frequently used cost function is the mean squared error given by

$$C(\theta) = C(\mathbf{y}, \tilde{\mathbf{y}}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \tilde{y}_i)^2.$$

Its derivative $D_{\mathbf{W}}$ w.r.t. the weights is

$$D_{\mathbf{W}} C(\theta) = -\left(\frac{2}{n} \sum_{i=1}^{n} (y_i - \tilde{y}_i) \right) D_{\mathbf{W}} \tilde{\mathbf{y}}$$

by the chain rule. Hence, with L^2 regularization, we have

$$D_{\mathbf{W}} C(\theta) = \left(-\frac{2}{n} \sum_{i=1}^{n} (y_i - \tilde{y}_i) \right) D_{\mathbf{W}} \tilde{\mathbf{y}} + 2\lambda \mathbf{W}$$

and for L^1 regularization,

$$D_{\mathbf{W}} C(\theta) = \left(-\frac{2}{n} \sum_{i=1}^{n} (y_i - \tilde{y}_i) \right) D_{\mathbf{W}} \tilde{\mathbf{y}} + \lambda \operatorname{sgn}(\mathbf{W}).$$

III. IMPLEMENTATION

A. Hyper parameter tuning

To tune the hyper parameter λ of the regularization, we use a random grid search in the regression problem. In the classification task, we only perform a simple grid search.

In the former case, we first sample four values out of the set $\{10^{-j} \mid j = 0, ..., 6\}$ look for the optimal λ among these values. Afterwards, we refine the grid around the current optimal lambda by sampling five more numbers from a logarithmic scale with 10 numbers, starting at 0.5λ and ending at 5λ. Then we look for the best hyper parameter among the new samples and compare it with the current λ. Finally, the hyper parameter is set to the one value out of the two which leads to the best MSE. On the one hand, this procedure is simple to implement, and the computational cost is lower than that of a simple deterministic grid search. On the other hand, the accuracy of this method is a bit lower.

To tune the learning rate η, we also perform a random search on a logarithmic scale with one refinement. However, we only do 10 epochs of the training since a good decay of the loss can be already observed after few epochs. This saves computation time.

B. Network training

Algorithm 1 Training a Neural Network with Mini-Batch Gradient Descent

Require: Training data $(\mathbf{X}, \mathbf{Y})$, number of batches B, optimizer $\mathcal{O}$, epochs E, cost function

Initialize optimizers for each layer's weights and biases
for $e = 1$ to E **do**
  $(\mathbf{X}', \mathbf{Y}') \leftarrow \text{Resample}(\mathbf{X}, \mathbf{Y})$
  Split $(\mathbf{X}', \mathbf{Y}')$ into B batches
  for each batch $(\mathbf{X}_b, \mathbf{Y}_b)$ **do**
    for each layer l **do**
      $W_l \leftarrow W_l - \mathcal{O}_{W_l}.\text{update}(\nabla W_l^b)$
      $b_l \leftarrow b_l - \mathcal{O}_{b_l}.\text{update}(\nabla b_l^b)$
    end for
  end for
end for
return $[(W_l, b_l)$ for l in layers$]$

We explain the training of our network with Algorithm 1. In the beginning all optimizers are initialized. For every layer we have one optimizer for the weights and one for the bias. Afterwards we start in every epoch to resample our data with replacement. This leads to duplicates and thus we have in every epoch a slightly different dataset which should help to avoid overfitting. Now we split the resampled data into batches. For every layer we perform one gradient step for the weights and the biases on the corresponding batch.

C. Data preprocessing

The MNIST Dataset consists of black and white images with 28 x 28 pixels. Each of these pixels is represented by a number from 0 to 255 where 255 stands for a complete white pixel and 0 for a complete black one. As our network can not handle matrices as input, we flatten the pixel matrix by stacking the rows. This leads to a small loss of spatial information. A further development of neural networks are Convolutional Neural Networks which can handle this aspect better. It is advantageous to scale the data, because this helps the gradients to stay in a reasonable range and thus helps to avoid the vanishing gradient problem. We decided to scale our data to a range from 0 to 1 and achieve this by simply dividing all values by 255.

The targets of our data are the numbers 0 - 9. As usual in classification tasks we predict the probability or a score for every class and use as prediction the class with the highest score. To do so, we use one hot encoding and represent a target with the value k as 10 dimensional vector with a one in the k-th entry and zeros in all the other entries.

IV. RESULTS

A. Comparison of back-propagation with Autograd

Before we start our numerical experiments, we want to check that our self-written code is correct. Therefore we compare the gradients computed with our own back-propagation code with the ones from Autograd [9] and the performance of our own model with one from PyTorch [10] of the same architecture. We did this with L^1 and L^2 regularization as well as without regularization as this leads to different gradients.

For the gradients we calculated the gradient of a model with two hidden layers, where the activation functions were ReLU and sigmoid and an output layer with the identity function. This was implemented as a pytest and can be found in the Github repository. For the testing of the whole fitting process, we used three models with Adam, RMSProp and plain gradient descent. As activation functions ReLU, Leaky ReLU and sigmoid were used in the hidden layers and the identity function for the output layer. Every test was made with 100 samples and 1,000 epochs. As data, the Runge and 2D Gaussian function were used. As results only similar and not the exact same results can be expected since we did not initialize the weights and biases. The results are in Table A.7, A.8 and A.9

B. Regression problems

First, we test our neural network on regression problems. To this end, we always use noisy data

$$y = f(x) + \varepsilon$$

in the following, where $f\colon \mathbb{R}^d \to \mathbb{R}, d \in \mathbb{N}$, and $\varepsilon \sim \mathcal{N}(0, 0.01)$. We approximate different functions, starting

with the one-dimensional Runge function from Project 1, defined as

$$f_1(x) := \frac{1}{1+25x^2}.$$

Later, we extend our numerical experiments by more sophisticated functions, e.g. the two-dimensional Runge function

$$f_2(x,y) := \frac{1}{1+(10x-5)^2+(10y-5)^2}$$

or the Rastrigin function [11] which is defined by

$$f_3(x,y) := 20 + x^2 - 10\cos(2\pi x) + y^2 - 10\cos(2\pi y).$$

The latter function is in fact frequently used to test optimization algorithm as it admits many local minima, but only one global maximum in the origin. However, due to its high oscillations it might also be harder to fit than, e.g., the Runge function.

The Rastrigin function can be defined in more general terms in d dimensions as

$$f(x_1,...,x_d) := Ad + \sum_{i=1}^{d} \left(x_i^2 - A\cos(2\pi x_i)\right) \qquad (9)$$

where $A \in \mathbb{R}$ is a constant, usually $A = 10$. In our experiments we work with the Rastrigin function in the cube $[-1,1]^d$.

We start the discussion of the results with the comparison of the use of different activation functions. First, we consider only the 1D Runge function, approximated by a neural network with two hidden layers with 50 nodes each. We use two networks with the exact same architecture, the same training and test data, but once with sigmoid as activation function in both layers and once with ReLU in the first and sigmoid in the second hidden layer. In the output layer, the activation function of both networks is the identity. This makes sense for regression problems because we do not want to distort the output. As optimizer we chose Adam and tuned the learning rate η accordingly.

Figure 3 shows the approximation of the 1D Runge function using sigmoid in both hidden layers. Here, the learning rate was set to $\eta = 0.0387$. The results are very accurate and we obtain a smooth curve. The test MSE in this example is 0.0109. The approximation obtained by the neural network with ReLU in the first hidden layer and sigmoid in the second one, seen in Figure 4, yields a minimal better test MSE, namely 0.0102. The learning rate was set to $\eta = 0.1$. However, the solution curve is not as smooth as the first one. Hence, using only sigmoid as activation function probably yields slightly better result since the shape of the approximating curve fits better to the original one. Also, the difference in the test MSEs is not significant and can change if we use different random data.

Next, we executed the same experiment with only one hidden layer and sigmoid as activation functions. If 50

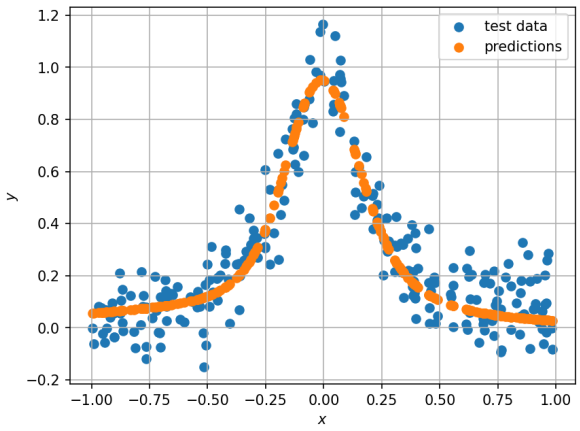

Figure 3. Approximation of the 1D Runge function with sigmoid in both hidden layers

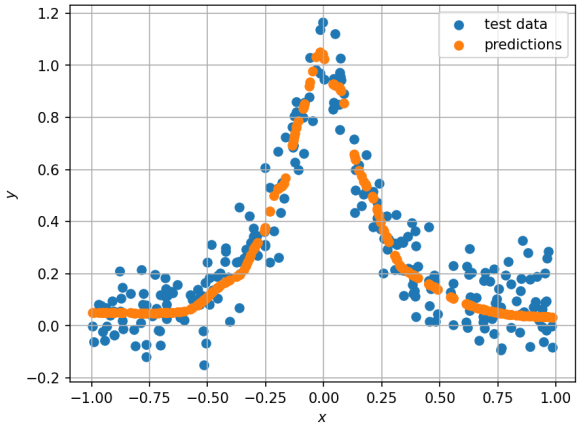

Figure 4. Approximation of the 1D Runge function with ReLU and sigmoid in the hidden layers

nodes are used, the test MSE was approximately 0.017 and with 100 nodes, approximately 0.022. Hence, the same number nodes, 100 in this case, leads to a smaller MSE, if the network has two layers instead of one. If we use two layers, both with 100 nodes, this does not improve the MSE either. We achieved a test MSE of 0.014 instead. Of course, the results vary from experiment to experiment, but we see that there are no significant improvements.

If we compare the results in Figures 3 and 4 with a polynomial fit, we also see that there are fewer oscillations toward the boundary of the interval $(-1,1)$ as the approximating function does not need to be a polynomial. Also, the test MSE is lower than the one of OLS (analytic or with gradient descent methods) to fit the function by a polynomial of low degree. Compared to OLS, the neural network only minimizes the difference between the approximating curve and the original data without constraints on the shape of the former. That is, we have more degrees of freedom that can be optimized. For higher polynomial degrees, the test MSEs are equal,

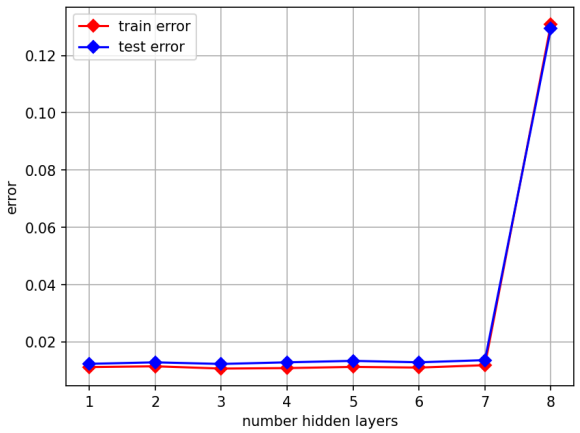

Figure 5. Train and test MSE for the 1D Runge function depending on the number of hidden layers. Each layer contains 50 nodes.

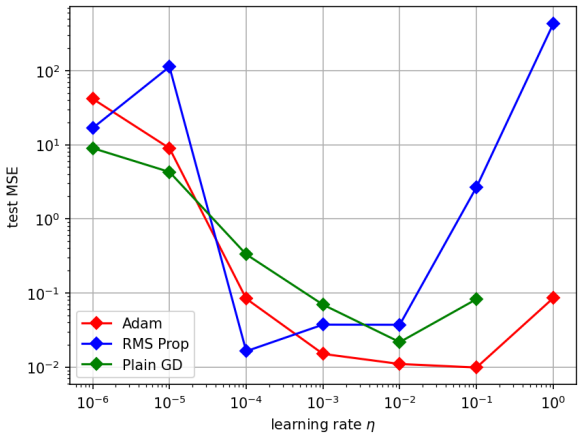

Figure 6. Test MSE for different optimizers and different learning rates

namely approximately 0.01. We obtained the best results for OLS when fitting the Runge function with polynomials of roughly degree 8 to 12. In fact, in both cases, the error is optimal, since it is the variance of the noise.

We can infer from these results that sigmoid as an activation function works best for this regression problem with our network architecture. A combination of ReLU and sigmoid is also suitable, in particular, if we use many hidden layers. However, if we use only the ReLU function in this example, we run into an overflow error in the backpropagation and obtain very poor results. In general, from our experiments we deduce that in no two consecutive layers, ReLU or LeakyReLU should be chosen as the activation function, given the same initialization of the network. For our network, we initialized the weights and biases by standard normal distributed random numbers, which yields a too large variance for ReLU. We found out too late that above-mentioned problems with ReLU do not arise when the weights and biases are initialized more suitably [12, 13]. In this case, networks using ReLU are even numerically more stable than the ones equipped with sigmoid.

In fact, using only sigmoid can lead to problems in the training for deep networks since it may cause the problem of vanishing gradients. The derivative of the sigmoid function, given in equation (8) takes only values between 0 and 0.25 and thus, the derivative become small very fast. Figure 5 demonstrates this effect. Here, we considered the approximation of the 1D Runge function, using neural networks with different dephts. For a too large number of hidden layer, the train and test error can become very high, it does not happen in all experiments, however. Nevertheless, we see that for a number of hidden layers between one and seven, we always achieve very good test MSEs, namely approximately 0.12. That is, in this case it makes sense to use neural networks with only a few hidden layers since the results are already satisfying and there is no need to increase the network depth.

We also tested our network in the setting where we keep the number of hidden layers, activation functions etc. fixed, but use different optimizers and different learning rates. Figure 6 shows the test MSE for Adam, RMSProp and plain gradient descent. Here, one observes that Adam performs best with a test MSE of approximately 0.01, but RMSProp with a suitable learning rates also achieves satisfying results. Plain gradient descent yields slightly worse results, but also reaches a test MSE of approximately 0.02 in the best case.

Our tuning of the learning rate yields $\eta_{\text{opt}} = 0.00387$ for Adam with a test MSE of 0.00926. For RMSProp we have $\eta_{\text{opt}} = 0.00108$ with a test MSE of 0.03448 and for plain gradient descent, the tuned learning rate is $\eta_{\text{opt}} = 0.02321$ leading to a test MSE of 0.01824. That is, we either actually achieve the best MSE or are close to it. In the latter case, the optimum is not achieved since we tuned the learning rate via random search.

In the tables A.2, A.3 and A.4, we also investigated the tuned leraning rate η_{opt} and corresponding test MSE for Adam, RMSProp and plain gradient descent, respectively, for different numbers of hidden layers. We see that for a larger number of hidden layers, we tend to need a higher learning rate for Adam and RMSProp. With both optimizers, we achieve a test MSE of approximately 0.01 in all cases. When we use plain gradient descent, we achieve better results with more hidden layers. For the learning rate we do not find a clear tendence here, however.

Next, we will briefly discuss the „Curse of dimensionality “. Most numerical methods perform best in low dimensions but are not suited for high-dimensional problems. This is for example also true for regression problems: While linear regression, as discussed in Project 1, works well for low-dimensional functions, the error grows rapidly with increasing dimensionality. Neural networks do not suffer from this problem. To demonstrate this, we tested our network on the d-dimensional Rastrigin function where $d = 2, ..., 5$. Since the computational work is

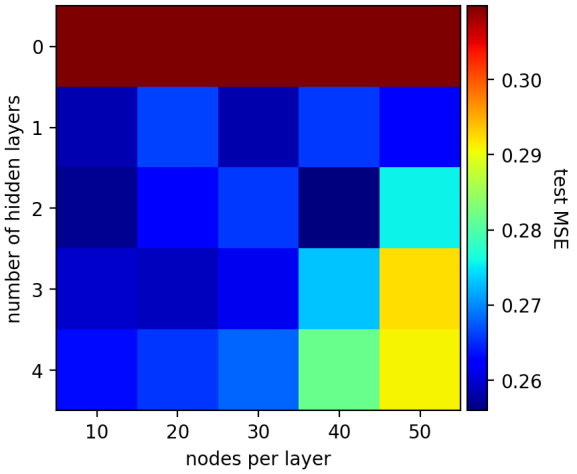

Figure 7. MSE of the approximation of the 1D-Runge function using neural networks with different numbers of hidden layers and nodes per layer and no regularization

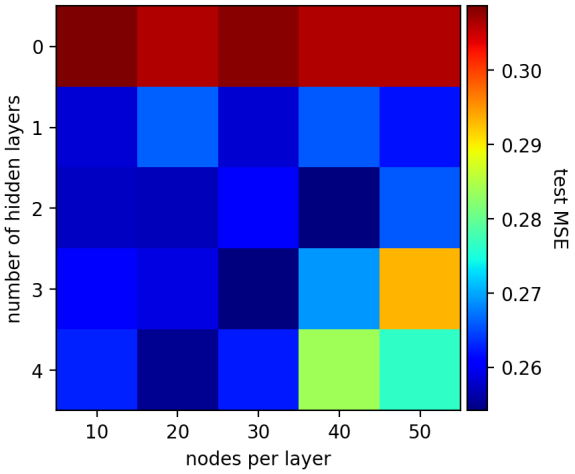

Figure 9. MSE of the approximation of the 1D-Runge function using neural networks with different numbers of hidden layers and nodes per layer and L^2 regularization

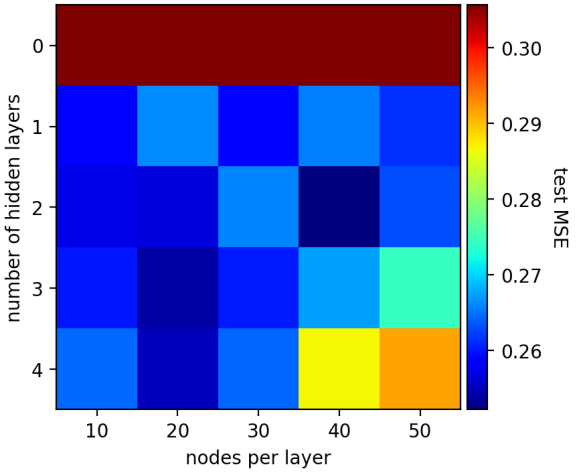

Figure 8. MSE of the approximation of the 1D-Runge function using neural networks with different numbers of hidden layers and nodes per layer and L^1 regularization

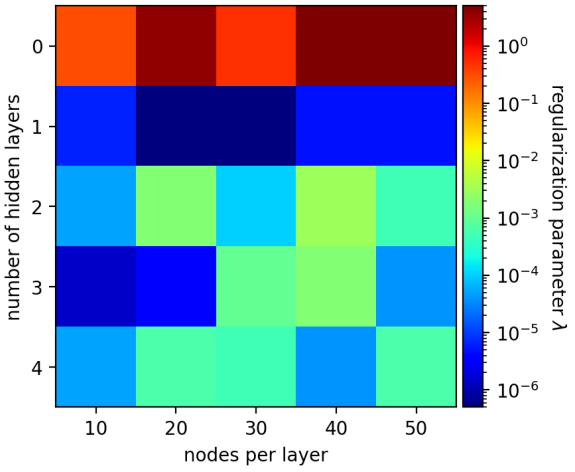

Figure 10. Hyper parameters λ of the L^2 regularization depending on the number of hidden layers and number of nodes per layer.

very high for this experiment, we only work with a simple network with one hidden layer with 50 nodes and sigmoid as activation function.

We used Adam with a tuned learning rate as optimizer. The results are shown in Table A.1. Here we see that the MSE does not grow too much with increasing dimension d. Of course we need more points for the training and more epochs in the back-propagation, which leads to a longer run time. Nevertheless, we do not need to change the network architecture for satisfying results. Moreover, we did not expect a network with such a simple architecture to achieve a test MSE of approximately 0.1 or lower in all examples.

To investigate the influence of regularization with the L^1 or L^2 norm on the approximation quality, we created neural networks with $i = 0, ..., 5$ hidden layers and $10j, j = 1, ..., 6$ nodes per layer. We did this three times, once without regularization and afterwards with L^1 and L^2 regularization. Thereby, we always tuned the hyper parameter λ accordingly. Then we computed the test MSE of those networks approximating the 1D Runge function in the interval $(-1, 1)$. In this particular case, we set the variance of the noise ε to 0.25. We used 750 points for training and 250 points for testing. As optimizer, we chose again Adam, where the learning rate was set to 0.01. We also used a combination of ReLU and sigmoid as activation functions, sometimes even ReLU in consecutive layers. This choice is intentionally not optimal for our network, as discussed above. But it allows us to see differences between the version without and the one with regularization. If we used our best model, this

would not be the case.

Figure 7 shows the test MSE when no regularization is used. One can see, that networks with more layers and more nodes per layer tend to yield a lower test MSE. However, for 4 or 5 hidden layers and 50 to 60 nodes per layer, we can observe signs of overfitting.

In Figure 8 we see that this effect is reduced a bit when we perform a regularization with the L^1 norm even though in the case of 4 hidden layers with 50 nodes per layer the test MSE becomes even higher. However, in total we achieve a small decay in the test MSE. Regularization with the L^2 norm, see Figure 9, yields roughly the same results. We see again that the test MSE is decreased in most cases. Nevertheless, in both cases, we do not obtain a drastic reduction of the test MSE. In Figure 10 we plotted the hyper parameters λ of the L^2 regularization that lead to the results in Figure 9. We observe that for higher numbers of hidden layers and nodes per layer the hyper parameter λ grows. This is expected since overfitting may appear in this setting.

For the simple example of approximating the Runge function, we can conclude that regularization does improve the test MSE for a high number of hidden layers and nodes per layer. But, firstly, for smaller numbers, we get equally good or better results without regularization. This resembles the observation from polynomial regression, that Ridge and LASSO regression do not improve approximations for low-complexity models and lead to the same results as OLS. In fact, a notable improvement only arises when we tend to overfit the data. Secondly, the differences between the L^1 and the L^2 norm are negligible in the considered case.

To end the subsection about regression problems, let us discuss the benefits and drawbacks of using neural network for approximation. As we have seen, a neural network can fit a continuous function with high accuracy, being better or as good as analytical or other numerical approaches, such as ordinary least squares (OLS), Ridge or LASSO. In contrast to polynomial fits, the solution obtained by the neural network does not admit oscillations toward the boundary of the interval.

In contrast to neural networks, a solution to OLS or Ridge can be found analytically and thus, these approaches do not suffer from numerical errors. Moreover, the computational work is higher for neural networks and hence, it seems to be too sophisticated to use one for a simple regression problem. Nevertheless, neural networks beat the curse of dimensionality and are therefore suitable for high-dimensional problems.

C. Classification

The first step in building a neural network is to find a good architecture. We had very decent results with funnel-shaped neural networks, i.e. the following hidden layer has half as many nodes as the previous. An example for the MNIST Dataset would be a neural network

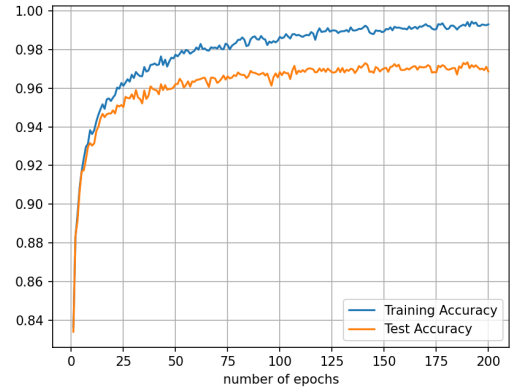

Figure 11. Test and Training Accuraccy during the training of a neural network

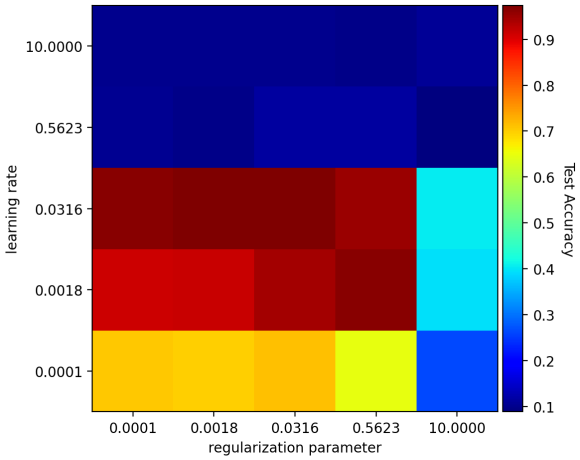

Figure 12. Test accuracy of a neural network with different learning rates and regularization parameter

with input layer 784, hidden layers 256, 128, 64 and output layer 10. We expect that this architecture gradually reduces model complexity and therefore forces the network to learn compact features. Although there are a lot of other architectures that work well on this data set, we find it fruitful to stick in the following to this architecture when we discuss various activation functions, number of hidden layers and nodes and hyperparameters. As Adam performed well in our Neural Network for Regression we will use it for Classification, too. We did not see remarkable differences between L^1 and L^2 regularization in the previous section and will stick to L^2. As we have a lot of different hyperparameters, it would be advantageous to find one learning rate that leads to decent results in various models. As higher batch sizes usually produce more stable gradients, these can afford higher learning rates. Therefore, we fix in the whole section the number of batches to 100 to justify the use of the same learning rate for all models. To find this learning rate we train a neural networks with hidden layers 128, 64 and 32 and

vary learning rate and L^2 regularization parameter. The results are presented as heatmap in Figure 12. We see that a learning rate of 0.03 performs best for almost all regularization parameters. This justifies to use in the following always the learning rate of 0.03. We want to discuss different activation functions and their order in the neural network. We train again a neural network with hidden layers 128, 64 and 32 and tune the learning rate. For a neural network which uses only sigmoid and for a network which uses ReLU sigmoid ReLU we get good results, i.e., 96.6 % and 97.4%, respectively, of test accuracy. The best performance is obtained by the model which uses ReLU sigmoid sigmoid with 97.6 %. Therefore we discuss in the following neural networks which have ReLU in the first hidden layer and sigmoid in the other hidden layers.

To get an understanding of the number of needed epochs we take a closer look at the training of a neural network with hidden layers 128 and 64. In Figure 11 we plot after every epoch the accuracy of our model on the training and the test data set. As expected, the training accuracy is monotonic increasing and converges almost to 100%. The test accuracy is always lower than the training accuracy and flattens faster than the train accuracy, but we do not see an effect that the test accuracy decreases after too many epochs due to overfitting. Therefore, we recommend to use about 100 epochs to obtain a good performing model in a reasonable training time.

For an analysis of the number of hidden nodes we look at a network with two hidden layers and let the number of hidden nodes vary from 512 and 256 to 32 and 16, respectively. As you may see in Table A.5 we observe the best accuracy for the highest number of hidden nodes. Using L^2 regularization improves every model, but we see the largest effect for the most complex ones.

To discuss the number of hidden layers we build the following setup, where we keep the number of nodes of the first layer constant and keep adding hidden layers with the number of nodes corresponding to the funnel-shaped architecture. This results in the smallest model with just one hidden layer with 128 nodes and the largest model with four hidden layers with nodes from 128 to 16. As in our previous example we can improve the performance of every model with L^2 regularization. We obtain the best results for four hidden layers but the test accuracies of the other models are very close.

Of all the models we tested the best one was the one with the most parameters: A neural network with hidden layers 512 and 256 with ReLU and sigmoid, 100 batches and epochs, Adam with a learning rate of 0.03 and an L^2 regularization term of 0.021544. To come up with these hyperparameters we have used 60 000 images to train and evaluate various neural networks. To test the model we use the remaining 10 000 images. On this dataset we achieve a remarkable accuracy of more than 97.8 %. Although the model predicts almost all images correctly, it is particularly interesting to have a closer look at the images that are classified wrong. Figure 13 shows the confusion matrix. As our dataset is large we have even without stratified sampling approximately the same number of samples per digit. We see that the most frequent misclassifications are five predicted as three and nine predicted as four. Even for humans it can be challenging to distinguish between these digits. As an example you can see in Figure 14 a sample of a three which can be mistaken as a five.

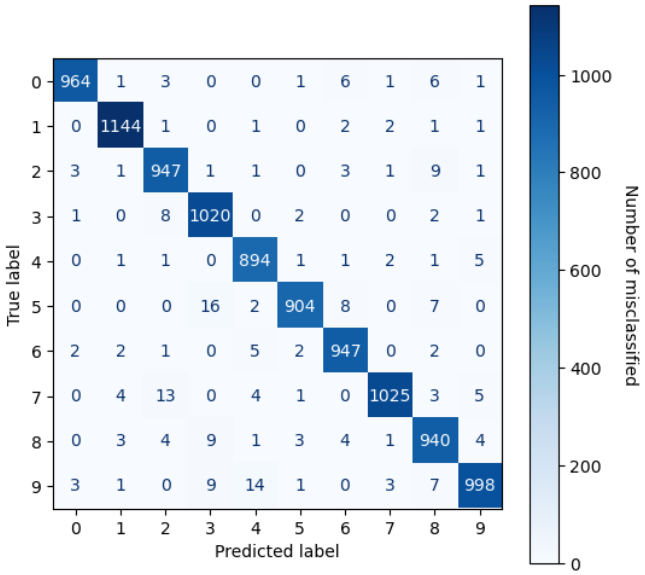

Figure 13. Confusion matrix of our best model

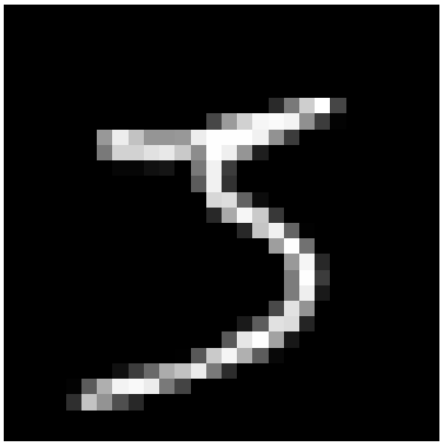

Figure 14. Sample from the MNIST Dataset: True value 3

V. CONCLUSION

In this Section, we want to briefly state our main findings. We start again with the analysis of the regression problem. In our experiments, we saw that a neural network with a small number of hidden layers can already

approximate a continuous function almost perfectly. This is consistent with the universal approximation theorem. In fact, one hidden layer is enough to reach a test MSE close to the variance of the noise in the data.

For our network architecture, sigmoid as activation function works best. However, with a different initialization of the weights, one should obtain more stable results with the ReLU function.

Comparing the performance of the neural network to other regression or approximation methods, namely OLS, Ridge and LASSO, we see that the test MSEs are optimal in both cases but using neural networks requires more computational work. A big advantage of neural networks compared to fitting a function with polynomials is that no oscillations arise at the boundary of the domain. Moreover, in higher dimensions, the neural network approximation remains stable and the test MSE does not grow significantly with increasing dimensionality.

In our classification task, the MNIST Dataset, we had very good results using a structure where the first hidden layer uses ReLU and the other hidden layers use sigmoid and every hidden layer has half of the nodes of the previous one. We found out that in this setting one specific learning rate works well for a variety of models. An interesting insight was that the test accuracy does not decrease after a certain number of epochs but stays constant even though the test accuracy converges close to 100 %. Thus, we do not overfit by using too many epochs. In our comparison of different model depth and number of hidden nodes, we saw a lot of models with very good performance. Even for simple models we saw an improvement by using regularization. In the end, the best model was the most complex one with hidden layers 512 and 256. We restricted ourselves to this model complexity, because our self-written neural network has a lower computational efficiency than libraries like tensorflow and pytorch. Furthermore, we sticked to run the whole code on budget-friendly laptops. For more complex models we expect even higher test accuracies.

[1] M. A. Nielsen, "Neural networks and deep learning," (2018).

[2] I. Goodfellow, Y. Bengio, and A. Courville, "Deep learning," (MIT Press, 2016) Chap. 8, http://www.deeplearningbook.org.

[3] G. Cybenko, Mathematics of control, signals and systems **2**, 303 (1989).

[4] K. Hornik, M. Stinchcombe, and H. White, Neural networks **2**, 359 (1989).

[5] J. F. Epperson, The American Mathematical Monthly **94**, 329 (1987).

[6] L. Deng, IEEE Signal Processing Magazine **29**, 141 (2012).

[7] S. An, M. Lee, S. Park, H. Yang, and J. So, "An ensemble of simple convolutional neural network models for mnist digit recognition," (2020), arXiv:2008.10400 [cs.CV].

[8] "Derivative of the softmax function and the categorical cross-entropy loss," https://medium.com/data-science/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1, accessed: 2025-11-05.

[9] D. Maclaurin, D. Duvenaud, and R. P. Adams, in *ICML 2015 AutoML Workshop*, Vol. 238 (2015) p. 5.

[10] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," (2019), arXiv:1912.01703 [cs.LG].

[11] H. Mühlenbein, M. Schomisch, and J. Born, Parallel computing **17**, 619 (1991).

[12] S. K. Kumar, "On weight initialization in deep neural networks," (2017), arXiv:1704.08863 [cs.LG].

[13] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," (2015), arXiv:1502.01852 [cs.CV].

Appendix A: Tables

d	train pts.	test pts.	epochs	η	test MSE	time [s]
2	750	250	500	0.2321	0.0312	1.500
3	7,500	2,500	750	0.8341	0.1150	16.199
4	37,500	12,500	1,000	0.1391	0.0220	219.816
5	75,000	25,000	1,500	0.6458	0.0814	734.869

Table A.1. Approximation of the d-dimensional Rastrigin function for different dimensions d using a neural network with only hidden layer with 50, sigmoid as activation functions and Adam as optimizer.

# hidden layers	η_{opt}	Test MSE
1	0.10000	0.01069
2	0.05000	0.01023
3	0.00500	0.01030
4	0.01077	0.01070
5	0.01000	0.01054

Table A.2. Optimal learning rate and test MSE using Adam with different numbers of hidden layers.

# hidden layers	η_{opt}	Test MSE
1	0.01000	0.01534
2	0.00050	0.01164
3	0.00834	0.01717
4	0.01000	0.02949
5	0.00083	0.01038

Table A.3. Optimal learning rate and test MSE using RMSProp with different numbers of hidden layers.

# hidden layers	η_{opt}	Test MSE
1	0.00387	0.03671
2	0.00500	0.02331
3	0.10000	0.01204
4	0.10772	0.01140
5	0.00500	0.01427

Table A.4. Optimal learning rate and test MSE using plain gradient descent with different numbers of hidden layers.

hidden layers	Test Accuracy
128	0.9732
128 64	0.9726
128 64 32	0.9736
128 64 32 16	0.9741

Table A.5. Test accuracys for neural networks with different number of hidden where the L^2 regularization was tuned

hidden layers	Test Accuracy (Tuned λ)	Test Acuraccy ($\lambda = 0$)
512 256	0.9774	0.9616
256 128	0.9757	0.9675
128 64	0.9753	0.9663
64 32	0.9757	0.9613
32 16	0.9686	0.9506

Table A.6. Test accuracys for neural networks with different number of hidden nodes and tuned L^2 learning rate vs no regularization

Own model	0.0113	0.0070	0.0102	0.0072	0.0113
PyTorch model	0.0108	0.0064	0.0091	0.0074	0.0113

Table A.7. Test MSE with Adam fitting the Runge function

Own model	0.0091	0.0097	0.0097	0.0159	0.0125
PyTorch model	0.0094	0.0101	0.0120	0.0149	0.0155

Table A.8. Test MSE with RMSprop fitting the Runge function

Own model	0.0113	0.0070	0.0102	0.0072	0.0113
PyTorch model	0.0108	0.0064	0.0091	0.0074	0.0113

Table A.9. Test MSE with constant learning rate fitting the 2D Gaussian function