Project 2 in FYS-STK4155

Adrian Martinsen Kleven
Simon Elias Schrader

Autumn 2020

Contents

List of Figures

List of Tables

1 Abstract

Feed forward neural networks (FFNNs) are among the most simplistic models for neural networks, but they have been implemented with incredible success in the past decades. They are also well documented, which makes them a safe playground for exploring their vast parameter space. We implemented a simple FFNN applied to regression of terrain data [5] and classification of handwritten digits (MNIST) [6] and compared those to OLS & Ridge regression and multinomial logistic regression. For the MNIST data set, we found parameters that gave a test accuracy of 99% using a single layer neural network with LeakyReLU activation function and ADAM stochastic gradient descent, compared to 97% using multinomial logistic regression with simple SGD. For the regression case, we got an MSE which was only 60% the MSE of OLS Regression using the tanh activation function and a two-layer Neural Network. Finally, we saw that our implementations of both multinomail logistic regression & the FFNN are on pair with the respective implementations in Scikit learn, yielding similar results. We also compared different stochastic gradient methods for linear regression and found that ADAM works well, however, it does not quite reach the error achieved using the analytical formulas.

2 Introduction

As can be seen in [5], both Ordinary Least Square (OLS) regression and Ridge regression failed to accurately fit a polynomial function to terrain data and did not manage to match the surface properly. In this article, we analyse whether regression with the help of feed forward neural networks (FFNNs) can give better results (in the form of a lower Mean Square Error) than OLS and Ridge regression. In order to do so, we implemented several stochastic gradient methods to find the approximate minimum of the Mean Square Error (MSE) function in parameter space. In order to evaluate their quality, we first compared their performance to the analytical expressions for OLS and Ridge regression for several parameters.
Later, these methods were used in the back propagation of the neural network. For the regression problem, we used the sigmoid function and the tanh function as well as ReLU and LeakyReLU as activation functions for the hidden layers and the linear function for the output layer. We did this comparison for several stochastic gradient methods and a flexible number of hidden layers and neurons per hidden layer.

We test and compare the performance of the neural network (equipped with the Softmax function on the output layer) on the classification of hand written digits using the MNIST data set [6]. We assess the effect of network architecture as well as the choice of hidden layer activation function.
Multinomial logistic regression is used to classify the same data set. We examine the effect of learning rate and the interplay between learning rate and the number of epochs.
Both the neural network and multinomial logistic regression is compared with the corresponding Scikit- Learn implementation in order to assess the success of our implementations. Finally we compare the use of our neural network and the multinomial logistic regressor applied to the classification of digits.

This report is structured so as to first introduce the relevant methods on a theoretical and conceptual level. After that, we cover some of the choices made in implementing these methods in code. In the Results section, we present the results of the code we implemented and comment on their implications. Finally, we share our conclusions about the results we've found.

All methods are implemented using the Python programming language. The programs are available at the Github address. Our programs were tested against relavant SciKit- Learn implementations [8].

3 Methods

3.1 Logistic Regression

Logistic regression is a regression algorithm applied to binary classification problems. A weighted and biased sum of predictors are passed through the logistic activation function and a cost function is applied to the output and the accompanying label to the set of predictors. The problem of finding the weights and biases that predict the correct category then becomes a problem of optimizing the cost function by tweaking the weights and biases. Logistic regression is in many ways comparable to linear regression. The key difference arises in the use of the Logistic or Softmax activation function.

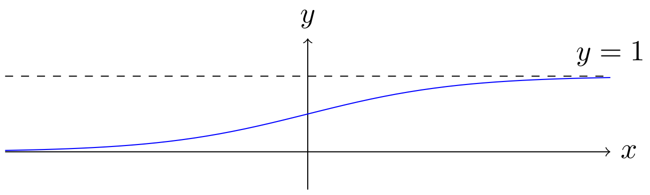

Figure 1: The logistic function outputs a number between 0 and 1 for every function input.

The logistic function, seen in figure 1 naturally translates a range of inputs into a probability distribution of two outcomes.

Multinomial logistic regression is the generalization of logistic regression to multi- class problems and uses the Softmax activation function. We will consider multinomial logistic regression from here on, as the Softmax function reduces to the logistic function in the case where the number of categories is 2 (7.1).

Given a dataset of L datapoints, N predictors and m categories, we structure the dataset thus

$$\mathbf{X} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,N} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,N} \\ \vdots & \vdots & \cdots & \vdots \\ x_{L,1} & x_{L,1} & \cdots & x_{L,N} \end{pmatrix} \quad \mathbf{Y} = \begin{pmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,m} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,m} \\ \vdots & \vdots & \cdots & \vdots \\ y_{L,1} & y_{L,1} & \cdots & y_{L,m} \end{pmatrix} \tag{1}$$

Where $\mathbf{X}$ is a $L \times N$ matrix with the predictors corresponding to a datapoint arranged along the row. $\mathbf{Y}$ a $L \times m$ matrix wherein every row of the matrix is a vector in the One- hot representation corresponding to the label describing the datapoint in the corresponding row in $\mathbf{X}$.
The weights are arranged in a $m \times N$ matrix and the biases in a $m \times 1$ vector

$$\mathbf{W} = \begin{pmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,N} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,N} \\ \vdots & \vdots & \cdots & \vdots \\ w_{m,1} & w_{m,1} & \cdots & w_{m,N} \end{pmatrix} , \; \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix} . \tag{2}$$

The weighted sum for a datapoint l is then given by

$$\mathbf{z}^{(l)} = \mathbf{W}\mathbf{x}^{(l)} + \mathbf{b} \; \textit{where} \; \mathbf{x}^{(l)} = (\mathbf{X}_{[l,:]})^T \tag{3}$$

and the $i'th$ activation by

$$a_i^{(l)} = \frac{e^{z_i^{(l)}}}{\sum_{j=1}^m e^{z_j^{(l)}}}. \tag{4}$$

Given that the labels are structured in the One- hot representation, the cross- entropy cost function for a single datapoint l is given by

$$C^{(l)} = \sum_{j=1}^{m} -y_j^{(l)} \log(a_i^{(l)}). \tag{5}$$

The algorithm can be understood as a single layer perceptron with m neurons in the hidden layer. In figure 2 below, the functions and variables in the nodes are as given in equations (1) to (5).

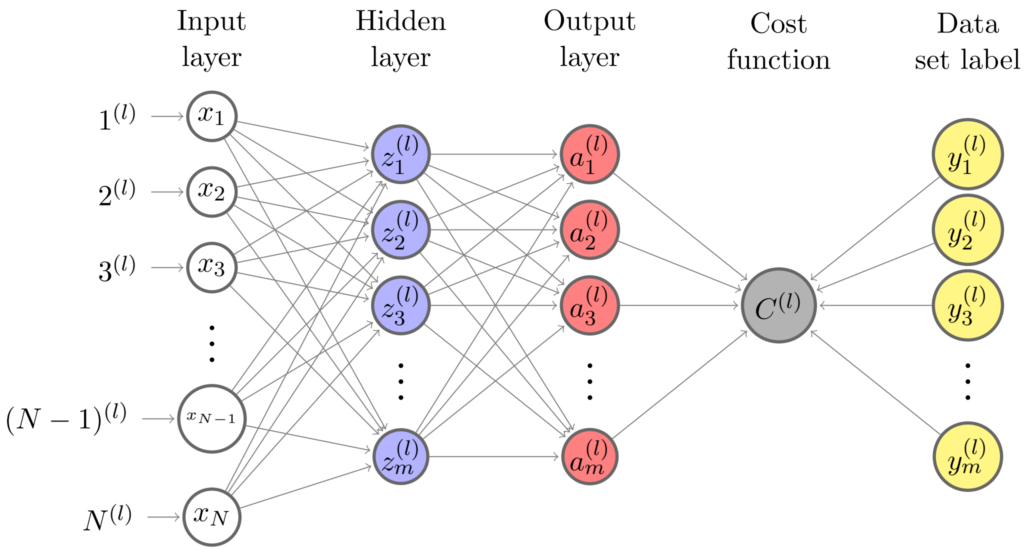

Figure 2: The single layer perceptron model of multinomial logistic regression.

3.1.1 Cost function

In stochastic gradient descent, the cost function we will end up using in the below calculations are given by

$$C = \sum_{l=1}^{Batch\,size} C^{(l)} \tag{6}$$

3.1.2 Derivatives of the cost function

We want to calculate the derivative of the new cost function with respect to the weights and biases. We will use these to calculate the step in the gradient descent: $\mathbf{W} = \mathbf{W} - \eta \frac{dC}{d\mathbf{W}}$ and $\mathbf{b} = \mathbf{b} - \eta \frac{dC}{d\mathbf{b}}$. Using the chain rule, we can re-express this problem as

$$\frac{dC}{dw_{i,n}} = \sum_{j=1}^{m} \left(\frac{dC}{da_j}\right)\left(\frac{da_j}{dz_i}\right)\left(\frac{dz_i}{dw_{i,n}}\right) \tag{7}$$

and

$$\frac{dC}{db_i} = \sum_{j=1}^{m} \left(\frac{dC}{da_j}\right)\left(\frac{da_j}{dz_i}\right)\left(\frac{dz_i}{db_i}\right) \tag{8}$$

where $i \in [1, 2, \cdots, m]$ and $n \in [1, 2, \cdots, N]$. We can now proceed to solving each of these derivatives in turn:

$$\frac{dC^{(l)}}{da_j^{(l)}} = -\frac{y_j^{(l)}}{a_j^{(l)}}, \quad \frac{dz_i^{(l)}}{dw_{i,n}^{(l)}} = x_n^{(l)}, \quad \frac{dz_i^{(l)}}{db_i^{(l)}} = 1. \tag{9}$$

The derivative of the Softmax function

$$\frac{da_j^{(l)}}{dz_i^{(l)}}$$

is

$$a_j^{(l)}(\delta_{ji} - a_i^{(l)})$$

but rather than this expression, we want to formulate the derivative as a $m \times m$ matrix:

$$\frac{d}{d\mathbf{z}^{(l)}}\mathbf{a}^{(l)} = (\mathbf{a}^{(l)} \otimes \mathbf{1}) \circ (\mathbf{I} - \mathbf{1} \otimes \mathbf{a}^{(l)}) = \mathbf{M} \tag{10}$$

where $\mathbf{1}$ is the $m \times 1$ one- vector, $\mathbf{I}$ is the $m \times m$ unit matrix, $\otimes$ is the outer product and $\circ$ is the Hadamard product. The reason for this seemingly convoluted change in perspective, becomes apparent when we consider our equations (7) and (8), which are now expressible (for a single datapoint) as

$$\frac{dC^{(l)}}{d\mathbf{W}} = \mathbf{M} \cdot (-\mathbf{y}^{(l)} \oslash \mathbf{a}^{(l)})\mathbf{x}^{(l)} \tag{11}$$

and

$$\frac{dC^{(l)}}{d\mathbf{b}} = \mathbf{M} \cdot (-\mathbf{y}^{(l)} \oslash \mathbf{a}^{(l)}) \tag{12}$$

where $\oslash$ is the Hadamard division. With the gradients in hand, we can now optimize the weights and biases through gradient descent.

3.1.3 Algorithm

The algorithm for Logistic regression goes as follows. We make an initial guess for the weights and biases. Using these and a data point we calculate the activation function. This is then used, together with the same data point in calculating the gradient of the cost function, which is used to calculate the new, optimized parameters $\mathbf{W}$ and $\mathbf{b}$.

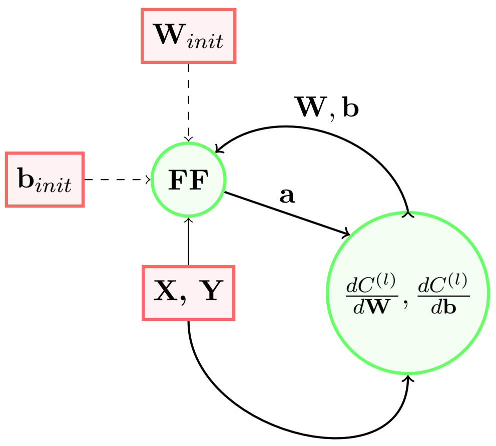

Figure 3: The algorithmic structure of logistic regression. Red squares represent variables that enter the algorithm. Green circles represent computations and dashed arrows means the output is used once.

This process continues until either a given number of iterations have taken place, or the Softmax function produces an output error within a given tolerance.

3.2 Gradient Descent Methods

One way to find the minima, both local and global, of a (multivariable) function, one can use the method of gradient descent. Simply speaking, this is done by iteratively changing the parameters in order to minimize a cost function [2]. As the gradient of a function always shows towards the point of steepest descent, following the gradient in the opposite direction will lead to a minimum. In both regression and classification problems, the function to be minimized is the cost function. In terms of linear regression, the cost function is the MSE function (possibly with additional regularization). The gradient is simply a vector containing the partial derivatives with respect to each coefficient β_i.
For OLS and Ridge regression, we have that

$$\nabla_\beta C(\beta) = \frac{2}{m}\left[X^T\left(X\beta - y\right) + \lambda\beta\right] \tag{13}$$

where C is the cost function, X is the design matrix and m is the number of inputs. The red part is only added for Ridge Regression.
After having an initial guess for the values β^0, The values β are then be updated iteratively by following the gradient in the opposite direction:

$$\beta^{i+1} = \beta^i - \gamma\nabla_\beta C(\beta^i) \tag{14}$$

where we introduced the learning rate γ. The learning rate γ needs to be chosen in such a way that it is not too large (which can lead to divergent behaviour), but not too small either (which can lead to an extremely slow convergence). This is done either until convergence is reached, or for a given number of iterations, called *epochs*.

3.2.1 Stochastic Gradient Descent Methods

Because calculating the gradient of every parameter β can be rather costly for large data sets, the gradient can be approximated by the gradient at only one input variable which is chosen randomly. This introduces randomness and erratic behaviour to the way the minimum is found. It is hence likely that the minimum is well approximated, but not exact [2]. However, the advantage is that the stochastic method can "jump out of" local minima and find the global minimum. One closely related method is Mini-batch gradient descent, where the gradient is approximated by the gradient at several, but not all, randomly chosen input variables. This leads to a less erratic behaviour, but is still computationally cheaper than Gradient Descent.
There are several ways to implement the actual gradient descent. It is useful to adapt the learning rate γ as the program proceeds - starting with a comparatively large learning rate, the algorithm can leave local minima and proceed to the global minimum, while the learning rate is gradually reduced to get better convergence. In the following, three methods of varying complexity will be introduced.

"Naive" Stochastic Gradient Descent has a constant learning rate γ, and the parameters β are just updated by (14) where the gradient is approximated. While this is easy to implement, has only one parameter (γ) to be fine tuned, and is cheap to calculate, the non-adaptive learning rate γ can lead to sub-optimal convergence.

Decaying γ - A simple way to make γ get smaller gradually is to implement a gradual decay. Defining

$$\gamma_t = \frac{t_0}{t_1 + t} \tag{15}$$

where t_0 and t_1 are initialization parameters and t is updated as $t = e \cdot m + i$ where e is the actual epoch, i is the actual mini batch and m is the number of mini batches. The update scheme remains the same (14), just that γ_t is used instead of a fixed γ. While this method has the advantage that γ_t gradually gets reduced, eventually γ gets so small that the steplength gets so small that no convergence is reached.

RMSProp describes a method where the learning rate is reduced gradually by accumulating the gradients from previous iterations, however, unlike the previous method, the impact of previous iterations decays exponentially. The update scheme is described as

$$\begin{aligned} \boldsymbol{s}^{i+1} &= \alpha \boldsymbol{s}^i + (1-\alpha)\nabla_\beta C(\boldsymbol{\beta^i}) * \nabla_\beta C(\boldsymbol{\beta^i}) \\ \boldsymbol{\beta}^{i+1} &= \boldsymbol{\beta}^i - \gamma C(\boldsymbol{\beta^i}) / \sqrt{\boldsymbol{s}^{i+1} + e} \end{aligned} \tag{16}$$

where * and / refer to element-wise multiplication and division, respectively. In this article, we chose the default value $\alpha = 0.9$, while $e = 10^{-8}$ simply has the purpose of avoiding zero division.

ADAM is a method related to RMSProp, and is a shortcut for *adaptive momentum estimation*. It combines features of RMSProp and a method called Momentum optimization (see for example [4]) - it keeps track of the average of past gradients, but also the average of past gradients squared, and both are to decay exponentially. The update scheme is described as

$$\begin{aligned} \boldsymbol{m}^{i+1} &= [\alpha_1 \boldsymbol{m} + (1-\alpha_1)\nabla_\beta C(\boldsymbol{\beta^i})](1-\alpha_1^T)^{-1} \\ \boldsymbol{s}^{i+1} &= \alpha_2 \boldsymbol{s}^i + (1-\alpha_2)\nabla_\beta C(\boldsymbol{\beta^i}) * \nabla_\beta C(\boldsymbol{\beta^i})(1-\alpha_2^T)^{-1} \\ \boldsymbol{\beta}^{i+1} &= \boldsymbol{\beta}^i - \gamma \boldsymbol{m}^{i+1} / \sqrt{\boldsymbol{s}^{i+1} + e} \end{aligned} \tag{17}$$

where * and / again refer to element-wise multiplication and division, respectively; and T stands for the number of iterations (starting at 1). In this article, we chose the default values $\alpha_1 = 0.9$ and $\alpha_2 = 0.99$, while $e = 10^{-8}$ simply has the purpose of avoiding zero division.

3.3 Neural networks

A Feed Forward Neural Network consists first of an input layer with neurons corresponding to each predictor in a data set. Then one or more hidden layers with neurons with weighted connections between every neuron in the two adjacent layers. Every neuron in the hidden layers are equipped with a bias, determining in a sense the activation threshold for that neuron. Additionally the weighted and biased sum of each neuron (with respect to the previous layer) is passed through an activation function. Finally, the activations reach the last hidden layer which is connected to the output layer, consisting of however many neurons needed to occupy the range of outputs we consider.

3.3.1 Back propagation

A neural network is parametrized by a set of Weight's and biases. When the cost function is being optimized, the weights and biases belonging to every neuron in the network have to be updated. The cost function only "sees" the activation from the output layer of the network. We therefore need to determine how the cost function changes with respect to the weights in all the hidden layers. Fortunately, we can accomplish this through the chain rule.

We have for the output layer of the network that through the chain rule we can write

$$\frac{\partial C}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathbf{z}^{(L)}}{\partial \mathbf{W}^{(L)}} \frac{\partial \mathbf{a}^{(L)}}{\partial \mathbf{z}^{(L)}} \circ \frac{\partial C}{\partial \mathbf{a}^{(L)}} \tag{18}$$

It's quick to check that $\frac{\partial \mathbf{z}^{(L)}}{\partial \mathbf{W}^{(L)}} = \mathbf{a}^{(L-1)}$. The remaining derivatives in the expression depend on the choice of activation function and cost function but are generally easy to calculate. We can also introduce the new variable $\delta^{(L)} = \frac{\partial \mathbf{a}^{(L)}}{\partial \mathbf{z}^{(L)}} \circ \frac{\partial C}{\partial \mathbf{a}^{(L)}}$. Equation (18) becomes

$$\frac{\partial C}{\partial \mathbf{W}^{(L)}} = \mathbf{a}^{(L-1)} \delta^{(L)}. \tag{19}$$

This is all well and good, but it doesn't tell us how we ought to change the weights in all the preceding layers. We'll introduce another expression, returning partially to equation (18), we see that can also, simply treat $\delta^{(l)} = \frac{\partial C}{\partial \mathbf{z}^{(l)}}$ rather than the Hadamard product. We'll use the chain rule again, but now to see how the cost function changes with respect to the succeeding layer:

$$\delta^{(l)} = \sum_k \frac{\partial C}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial \mathbf{z}^{(l)}} \tag{20}$$

This gives an equation that can be iterated all the way from the last to the first hidden layer, and we can use it to update the weights and biases (the bias follows similarly).

3.3.2 Feed Forward

After every new weight and bias in the network has been calculated, it's simply a matter of calculating the activations belonging to each layer from the first to the last layer in the network (since the activation of one layer depends on the activations in the preceding layer). After this process the network has a slightly altered cost function to optimize and the process repeats.

3.3.3 Activation functions

As described in the previous to subsections, the activation function is one of the core elements in Neural Networks. In this project, we implemented the following four activation functions for activation between the input layer and the first hidden layer as well as between all hidden layers.

sigmoid function - The sigmoid function, defined as $\sigma(x) = \frac{e^x}{e^x - 1}$, outputs function values between 0 and 1 for all inputs.

tanh - The tanh function, which can be expressed in terms of the sigmoid function $tanh(x) = 2\sigma(2x) - 1$, can take functions between -1 and 1. Unlike the sigmoid function, it maps negative inputs to negative function values, while 0 is mapped to 0.

ReLU -The ReLU function (*Rectified Linear Unit*) is defined as $ReLU(x) = x^+$ (x if x is positive, zero otherwise). Unlike the tanh function and the sigmoid function, it does not suffer from vanishing gradients when the input values are large. It has been shown [**?**] that rectifiers can give better results in Machine learning, especially depper networks, than the sigmoidal functions.

LeakyReLU - The LeakyReLU function is defined as $LeakyReLU(x) = max(x, \alpha x)$ where $\alpha = 0.01$ (though other values are possible, too). Positive values are hence mapped to themselves, whereas negative input values are mapped to αx. Unlike the ReLU function, it has nonzero output for negative input values too, and the gradient vanishes nowhere, improving the problem of "dying" neurons.

For the output, we used no activation function (Regression) or the Softmax activation function (Classification).

Softmax function - The Softmax function defined in equation (4) is a generalization of the Logistic function (sigmoid), and is used to classify more than two categories.

3.4 Data sets

3.4.1 Regression: Terrain Data

For the regression analysis, we used the same data as in [5] - a black-and white image with resolution 3601×3601 pixel which represents an area in the Taebaek Mountains in South Korea with a total surface area of $3601 \times 3601 km^2$, hence each square kilometer is represented as one pixel, where colour intensity represents the height (black equals height at sea level).

3.4.2 Classification: MNIST data set

For classification, we use the MNIST data set, a data set consisting of 70.000 handwritten digits between 0 and 9, represented as a picture with resolution 28×28 pixel [6]. In this article however we used Scikit Learn's variant of the MNIST data set, which consists of 1797 elements with resolution 8×8 pixel.

4 Computational implementation

4.1 Multinomial logistic regression

We have chosen classification of handwritten digits in an 8×8 grid of pixels, as the classification problem. To implement this, the $(8 \times 8) \times L$ pixel data is flattened into a $(L \times 64)$ matrix with L being the number of datapoints. The image labels, originally a $(L \times 1)$ array of digits 0 through 9, is converted into a One- hot matrix representation of shape $(L \times 10)$. The computational implementation of Softmax regression follows from the illustrations in figures 2 and 3.

4.1.1 Numerically stable variants

The Softmax function defined in (4), is inherently prone to overflow. We can resolve this by making a small change:

$$a_i = \frac{e^{z_i}}{\sum_{j=1}^{m} e^{z_j}} \cdot \frac{e^{-c}}{e^{-c}} = \frac{e^{z_i - c}}{\sum_{j=1}^{m} e^{z_j - c}} \tag{21}$$

where we choose $c = \max\{z_i : i \in [1, 2, \cdots, m]\}$. In this way we ensure that the function does not overflow, as the exponent is always smaller than zero. Only a unique set of circumstances could cause this expression to be numerically unstable, for example if the sum in the denominator is ever zero.
Additionally, the Hadamard division used to calculate the gradients in equations (11) and (12) receives a tiny addition of 10^{-15} in the denominator to avoid division by zero.

4.2 Neural network

4.2.1 Numerically stable variants

We used the same Softmax function as for multinomial logistic regression. For the sigmoid function, we used Scipy's [10] expit function.

4.2.2 Setting up weights and biases for the neural network

As there is no clear rule how to set up weights and biases, other than that they should be initialized with a non-zero value, we first tried to set up the weights with a mean zero normal distribution with a small standard deviation $\sigma \approx 0.01$. However, we found that this yielded undesirable results, which made that especially ReLU and LeakyReLU gave unpredictable behaviour where the activation function gave very high numbers, eventually leading to numerical instability and overflow. Hence, we decided to follow the approach described in [3], where the weights are initialized randomly, following a mean zero normal distribution with standard deviation $\sigma = \sqrt{2/n\_inputs}$ where n_inputs here refers to the batch size. The biases were simply initialized with a small nonzero number - 0.001.

4.2.3 Functionality of the Neural Network

We designed a flexible Neural Network that works with any amount of hidden layers and any amount of neurons per hidden layer. It works with both classification and regression, using the softmax function as activation function for the output layer for classification, and simply the linear function $f(x) = x$ for the regression case.

4.2.4 Adapting the Neural Network for Classification

Very little is needed to apply the neural network in classification problems. The dataset needs to be flattened (For this type of neural network, other types may exploit patterns that are destroyed in this process) such that each datapoint can enter the network as a vector. The labels should be converted (if not already) to the corresponding One- Hot representation.
Beyond this, the output layer of the network needs to be equipped with either the Sigmoid- or Softmax activation function. There is always the choice of cost function which, invariably for our purposes should be the Cross- Entropy cost function.

4.3 Scaling

For scaling we used Scikit-Learn's StandardScaler. In the linear regression problems, all of the dataset was scaled. In Multinomial logistic regression, the predictors were scaled but the labels were not. None of the data used in the classifying neural network was scaled with the exception of the benchmarking code.

4.4 Test functions and reproducibility

In order to ascertain that our methods were working, we implemented test functions for SGD with OLS and Ridge as well as the neural network where, given simple training data with known results, the training data needed to given predictions within a relatively large tolerance.

4.5 Explanation of Code

All programs can be found on our Github address. In the main folder, an explanation of the code can be found with an overview of the folder structure.

5 Results

5.1 Comparison of SGD methods for OLS

Figure 4 shows, for a given OLS problem, the test MSE as a function of the learning rate η for a fixed number of epochs & a fixed batch size; the test MSE as a function of the number of epochs for a fixed learning rate & a fixed batch size; and the test MSE as a function of batch size for a fixed number of epochs and a fixed learning rate.

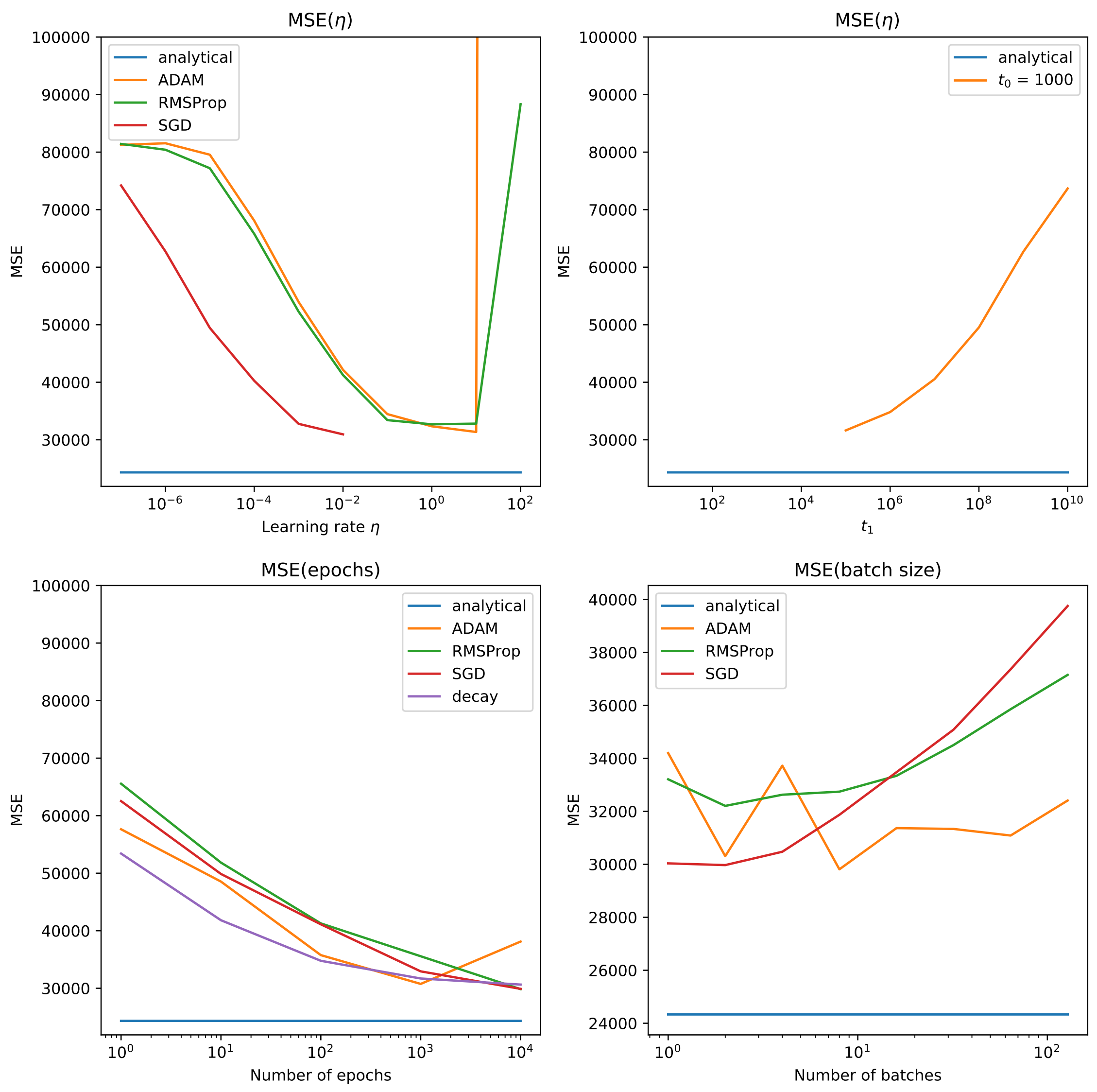

Figure 4: The simple SGD method (titled SGD), RMSProp, ADAM and decaying η as functions of the learning rate η (top left), the number t_1 (top right), the number of epochs (down left), and different batch sizes (down right). The number of data points is $N = 2000$, the polynomial degree used is $deg = 10$. For the top two plots, a batch size of 16 and and epoch of 1000 were chosen. The lower to plots use the ideal parameters η and t_1 which were chosen based on the ideal values from first two plots. No bootstrapping or cross-validation was performed.

As one can see, the number of epochs and the learning rate make a huge difference when it comes to approximating the analytical solution. For the decay-SGD and the simple SGD (titled SGD), the curves are truncated because too big or too small values lead to NaN-values. This shows that the ideal learning rate is dangerously close to a too high learning rate, leading

to completely wrong numbers or even NaN-values. Similar observations can be done for both ADAM and RMSProp, but the change is not as drastic for these methods.
As expected, the number of epochs lead to increased error reduction for all methods. However, even though the number of epochs grows exponentially, the error reduction slows down and even ceases. This is hence a computationally expensive way of reducing the extra error. As the learning rate was chosen to be ideal for 1000 epochs, we also see that, at least with ADAM, the error actually increases - this might be due to over-fitting, or leaving the reached minimum.
The number of batches does not seem to have a large impact on the quality of the fit for ADAM, but we observe that the simple SGD method and RMSProp work best with small batch sizes. This might be because these methods work best when making many "small hops" instead of several larger hops.
One can see that the choice of method has a large impact on how fast the error is reduced. RMSProp and ADAM seem to be slightly superior to the simple SGD in terms of convergence to the true MSE, however their biggest advantage is that they are more stable and have "broader" ideal learning rates. This is not surprising as these methods were developed for this purpose. ADAM seems to be better at dealing with higher epochs than RMSProp though. The decay-fit method, while giving results as good as RMSProp and ADAM for ideal parameters, is too unstable to be used in practice - small fluctuations in the parameters lead to completely wrong values. It is also harder to tweak several parameters. Figure 5 contains the same plots as figure 4, however, the learning rates η were chosen so that they didn't exceed a value of 0.1. This is more difficult to do for the decay method, which we left unchanged. One can see that this leads to a slightly different behaviour. The convergence is, not surprisingly, slower, but the methods behave slightly less erratic. That way, the error keeps reducing as the number of epochs increases, but it takes more epochs to get it to the same level as before. Also, the error now increases for larger batch sizes for all methods, including ADAM. In this implementation, a larger batch size has no computational advantages, however, for the Neural Network later, larger batch sizes give increased run time.

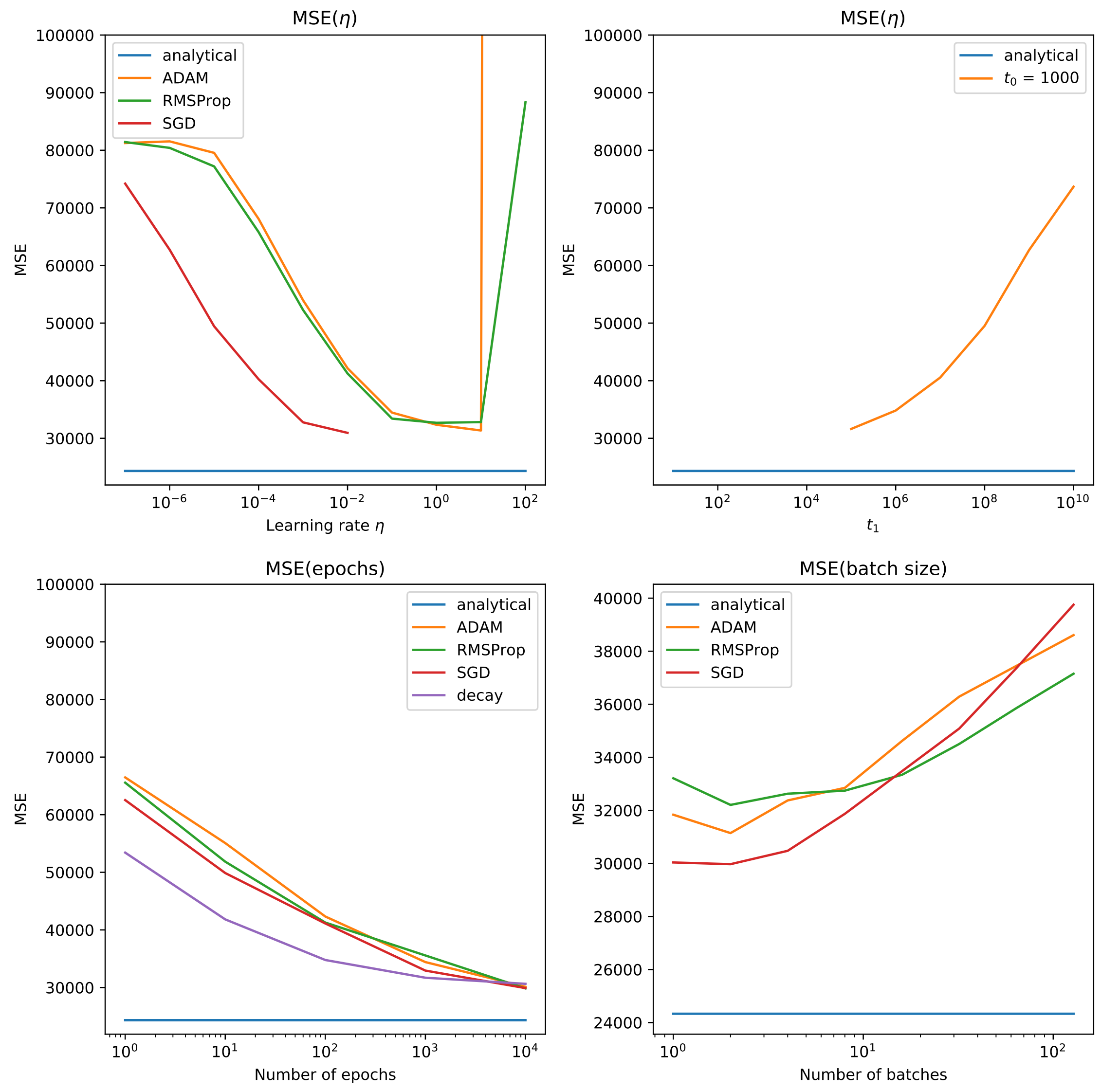

Figure 5: The simple SGD method (titled SGD), RMSProp, ADAM and decaying η as functions of the learning rate η (top left), the number t_1 (top right), the number of epochs (down left), and different batch sizes (down right). The number of data points is $N = 2000$, the polynomial degree used is $deg = 10$. For the top two plots, a batch size of 16 and and epoch of 1000 were chosen. The lower to plots use the ideal parameters η and t_1 which were chosen based on the first two plots, however, η was chosen so that $\eta \leq 0.1$ as larger numbers lead to instability later on. No bootstrapping or cross-validation was performed.

5.1.1 Comparison of SGD methods for Ridge regression

We repeated the same analysis as above with Ridge regression, only varying the learning rate and the regularisation parameter, keeping the batchsize fixed (16), as well as the number of epochs (1000). We chose a high polynomial degree where OLS is inferior to Ridge regression. The results can be seen in figure 16 in the appendix. The difference between the methods is baffling. We see that simple SGD gives NaN-values for too high learning rates, as before. RMSProp and ADAM, too, give worse results as the learning rate increases, but to a much lesser degree than simple SGD. As we did not perform Cross Validation, these numbers are only qualitatively correct, but we see that all methods, given the ideal parameters are chosen, can get very close to the analytical result. ADAM performs best and manages to come close to the analytical solution, however, both RMSProp and the simple SGD method get quite close, too. We see that regularization gives improved values for Stochastic Gradient Descent methods, to, as very small regularization parameters λ yield worse test errors than the optimal parameters. We see however that the error is always larger than the ideal test error, implying that Stochastic Gradient Descent methods can get quite close, but not exactly equal to the ideal analytical parameters, at least not with the chosen parameters.

5.2 FFNN for Regression

We used randomly selected points from the terrain data [5] to create a fit using both OLS and Ridge Regression, as well as the Neural Network.

5.2.1 Comparing the FFNN to Scikit-learn

We tested the quality of our Neural Network with N=2000 randomly selected data points and a polynomial degree of 10. We chose 200 as batch size and 1000 epochs. We used one hidden layer with 100 neurons. The sigmoid function was used as activation function for the hidden layer, and simple gradient descent was used. Figure 6 plots the test and train error as function of the regularization parameter λ and the error learning rate η. Using OLS, we found 28272 for the test MSE and 24758 for the train MSE, for comparison. We also compared this to Scikit-learn 's MLPRegressor function [8], which are included in figure 6.

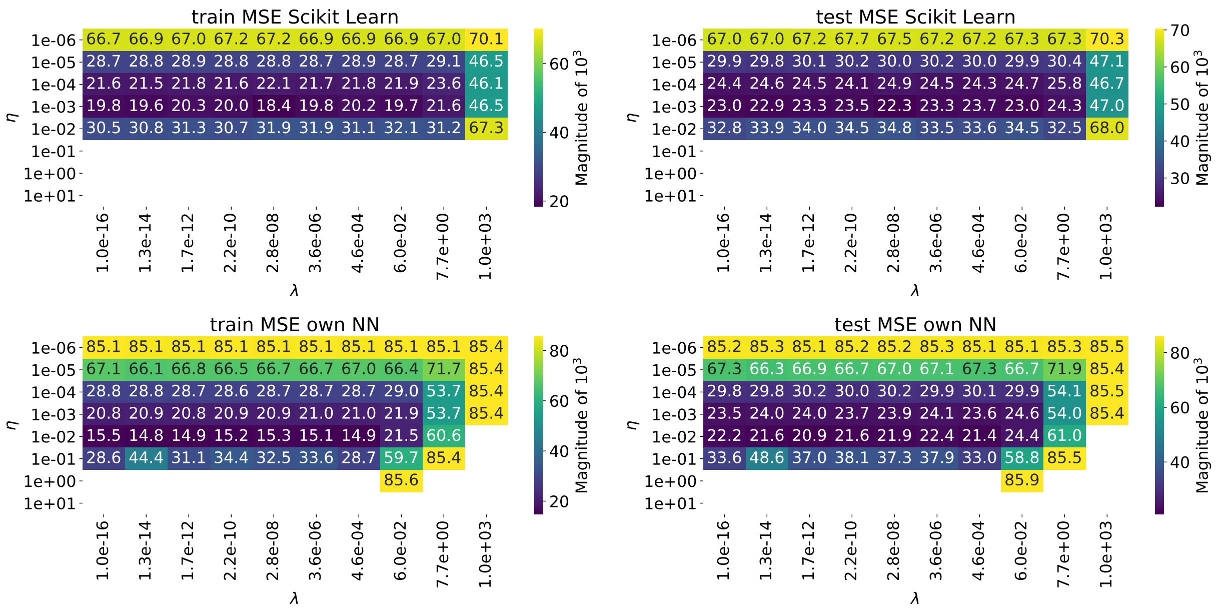

Figure 6: Train and test MSE <u>divided by 1000</u> as function of the learning rate η and the regularization parameter λ for our own Neural Network and Scikit-learn 's MLPRegressor function. We used 2000 data points (randomly selected), a polynomial degree of 10, a batch size of 200, 1000 epochs, one hidden layer with 100 neurons, simple SGD as gradient descent method and the sigmoid function as activation function between the layers. Values exceeding 100,000 are excluded from the plot. We used 5-fold Cross Validation to estimate the errors.

First of all, we see that both Scikit-learn and our own Neural Network outperform OLS (and Ridge regression, which gives identical values here). Values like that were not possible to obtain only using Linear Regression for that amount of data points [5], indicating that Neural Networks can give superior results to Linear Regression methods. This comes however at the

cost of not obtaining a nice function expression (it is up to the reader to decide if a multivariate polynomial of degree 10 is a nice function expression - but the number of parameters is bearable) with a meaning behind it.
We see that our algorithm gave superior values to Scikit-learn. This is supposedly due to a different implementation of the SGD-algorithm, which is clear given that the ideal parameters for the learning rate differ by one magnitude. Finally, we see that choosing wrong parameters ends up giving completely horrendous results, meaning that tweaking both the regularization parameter and the learning rate is necessary.

5.2.2 Impact of number of hidden layers

Exactly the same analysis as before (compare figure 6) was done, this time using two hidden layers with 100 neurons each. The results can be seen in figure 7.

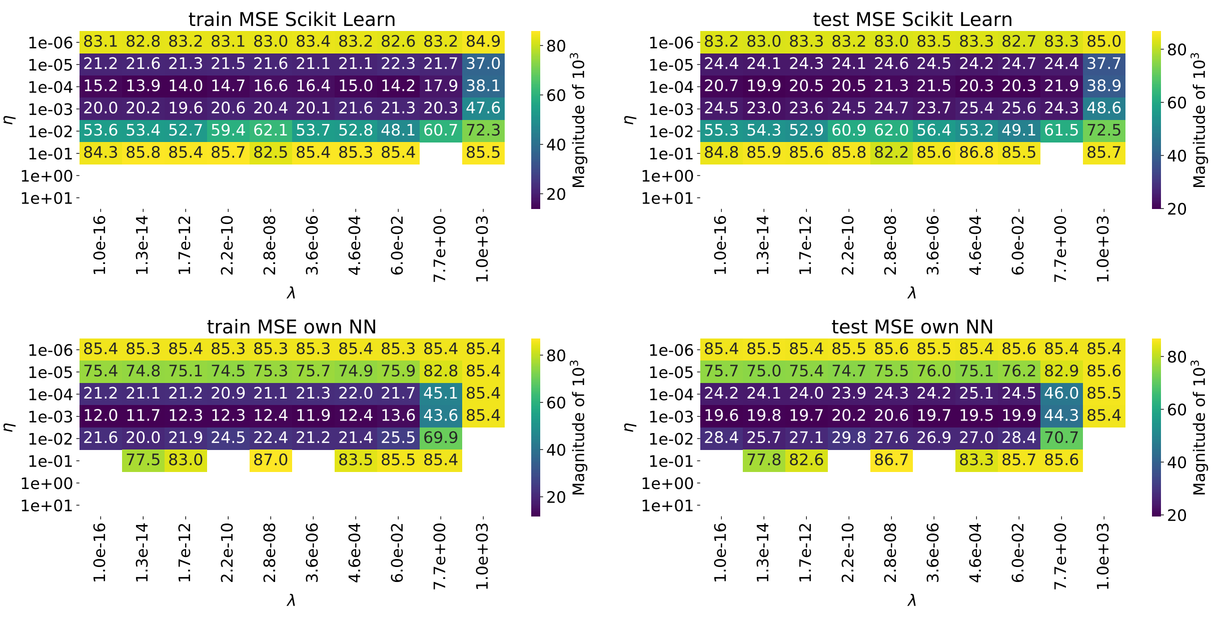

Figure 7: Train and test MSE divided by 1000 as function of the learning rate η and the regularization parameter λ for our own Neural Network and Scikit-learn 's MLPRegressor function. We used 2000 data points (randomly selected), a polynomial degree of 10, a batch size of 200, 1000 epochs, two hidden layers with 100 neurons each, simple SGD as gradient descent method and the sigmoid function as activation function between the layers. Values exceeding 100,000 are excluded from the plot. We used 5-fold Cross Validation to estimate the errors.

We see that adding the second layer gives even better results than just using one hidden layer. This indicates that using more layers can further reduce the error. Again, our Neural Network outperforms Scikit learn, but the difference is smaller than with just one single layer. We also run the same analysis with a polynomial degree of 20. The results can be seen in figure 17 in the appendix. Here, OLS fails due to a too high variance. Ridge regression can be used though and gave a test error of 24312, while our own Neural Network with 2 layers gave a test error of 20251 (which is slightly worse than the error produced by Scikit Learn, which is 20043. The test error for the neural network has hence increased - we suppose that the increased amount of parameters leads to a slower convergence rate, it might be possible that increasing the number of iterations might lead to better parameters.

5.2.3 Comparison between ADAM, RMSProp & simple SGD

Figure 8 contains the train and the test MSE using ADAM & RMSProp as stochastic gradient descent methods using our own FFNN. The parameters are identical to the ones in figure 7, that is, two hidden layers with 100 neurons each.

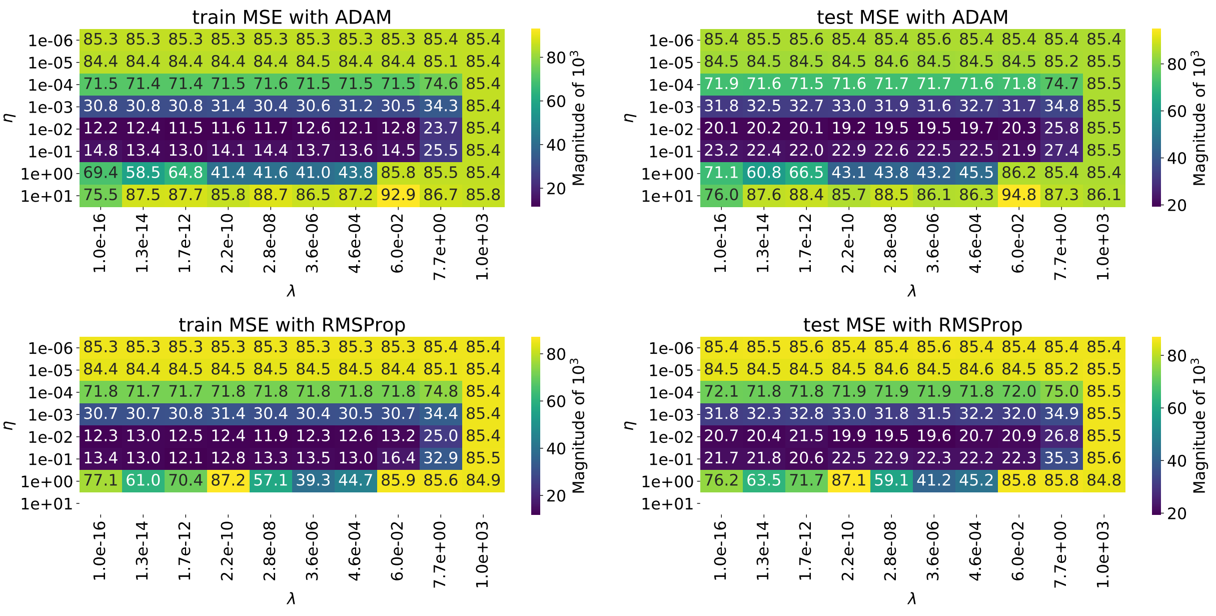

Figure 8: Train and test MSE divided by 1000 as function of the learning rate η and the regularization parameter λ for our own Neural Network using ADAM and RMSProp. We used 2000 data points (randomly selected), a polynomial degree of 10, a batch size of 200, 1000 epochs, two hidden layers with 100 neurons each and the sigmoid function as activation function between the layers. Values exceeding 100,000 are excluded from the plot. We used 5-fold Cross Validation to estimate the errors.

We see that the methods are generally similar and give similar test and train errors for similar parameters. However, ADAM copes better with too high learning rates and the results at high learning rates, albeit much worse than with good parameters, do not explode. ADAM also surpasses simple SGD in the sense that the lowest obtained test MSE is slightly lower, while RMSProp is on pair.

5.2.4 Impact of activation function

As ADAM has given the best test results, we used ADAM as stochastic gradient method and compared the impact of the four different activation functions ReLU, tanh, sigmoid & LeakyReLU. This can be seen in figure 9.

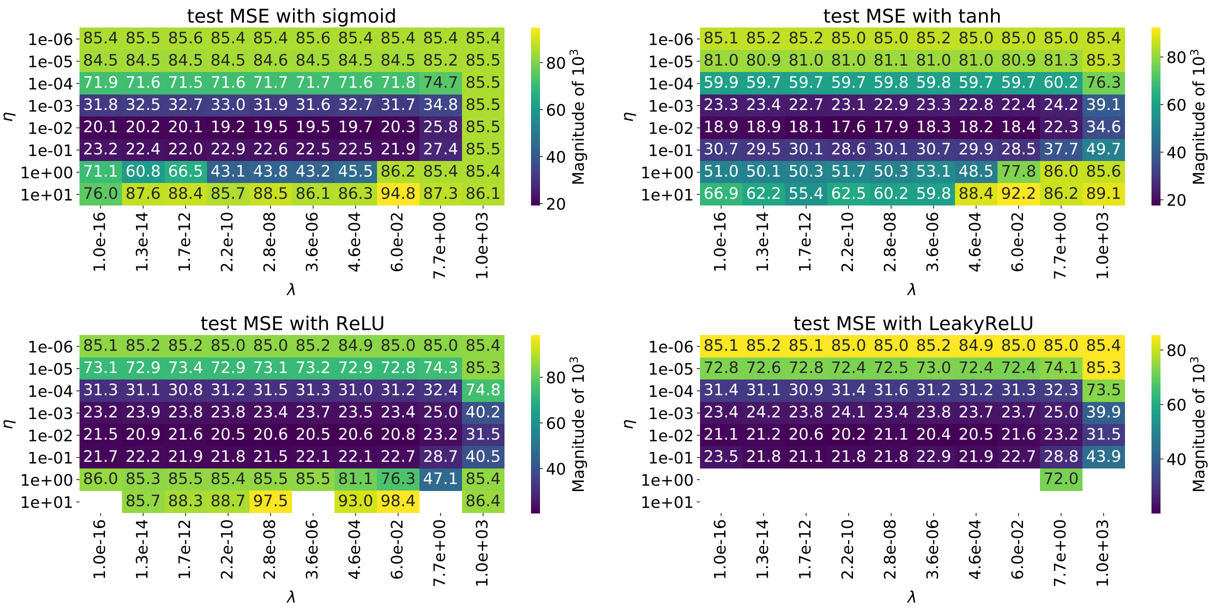

Figure 9: Test MSE divided by 1000 as function of the learning rate η and the regularization parameter λ for our own Neural Network using ADAM. We used 2000 data points (randomly selected), a polynomial degree of 10, a batch size of 200, 1000 epochs, two hidden layers with 100 neurons each and the activation function stated in the title between the layers. Values exceeding 100,000 are excluded from the plot. We used 5-fold Cross Validation to estimate the errors.

As we see, the tanh function gave by far the best results with a test MSE below $18,000$, followed by the sigmoid function. This does not show that the tanh function is the best activation function, but that the tanh function in this case, with the given amount of layers and chosen parameters as well as the number of epochs, performs best. It is interesting to see that both the ReLU and the LeakyReLU activaton functions seem to be more "forgiving" to different learning rates and regularization parameters, as a larger part of the 2D table appears blue, indicating small values, than with the tanh function & the sigmoid function.

For comparison, we run a small test with 4 hidden layers of sizes (100,100,50,50) (in that order, where the output layer is to the right), but only 100 epochs due to the high time usage. The result can be found in figure 18 in the appendix where the test error is portrayed for all 4 methods. From this graph, we see that ReLU & LeakyReLU perform much better for this deeper neural network. Not only is the smallest test error lower, the methods are much more stable too, in the sense that the both ReLU and LeakyReLU give good results for much more values of the learning rate and the regularization parameter, whereas tanh and especially the sigmoid function are very narrow. Figure 19 in the appendix is also run with only 100 epochs, but has only 2 hidden layers. Here, the sigmoid function and the tanh function are superior, however the test error achieved with both the LeakyReLU and ReLU and 4 layers is lower than the test error from the tanh function and two layers (and all sigmoid function values except for one), indicating that the ReLU/LeakyReLU can indeed give better results.

5.3 FFNN for Digit Classification

5.3.1 Comparing the FFNN to Scikit- learn

For comparison, we chose the Scikit Learn MLPClassifier. Parametrized with the ADAM solver using $\beta_1 = 0.9$ and $\beta_2 = 0.99$, an architecture of [100, 100, 50, 50], 100 epochs, a batch size of 100, learning rate of 0.001 and L2 regularization parameter of 0.0001. This is as close as possible to our own implementation using the ReLU solver. Both results we found using 4- fold cross validation. The results of our implementation can be found in figure 12. Using the parameters above, this model accomplishes a testing accuracy of 97%. The equivalent MLPClassifier function accomplishes a testing accuracy of 96%. Within the scope of random deviation, these results are essentially the same. This is a good indication that we have correctly implemented the Neural network and applied it to classification.

5.3.2 Impact of Network Architecture

To examine the impact that the choice of network architecture makes, we decided on four archetypes. Two variations on the single layer perceptron model, to see the impact of increasing the number of neurons. Additionally, a more square architecture and what I'll decide to call a thin and deep architecture. All of which are shown in figure 10 below.

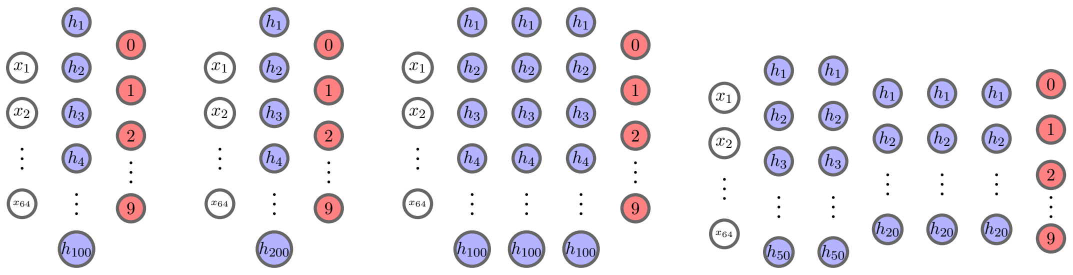

Figure 10: Four different NN- architectures. From left to right, we'll call them *Shallow1*, *Shallow2*, *Deep1* and *Deep2*. White nodes indicate the 64 input neurons. Blue nodes indicate hidden layer- neurons. Red nodes indicate the 10 output neurons.

Each of these were trained using the same parameters and their accuracy on test data measured using 4- fold Cross Validation. The results are shown in figure 11 below.

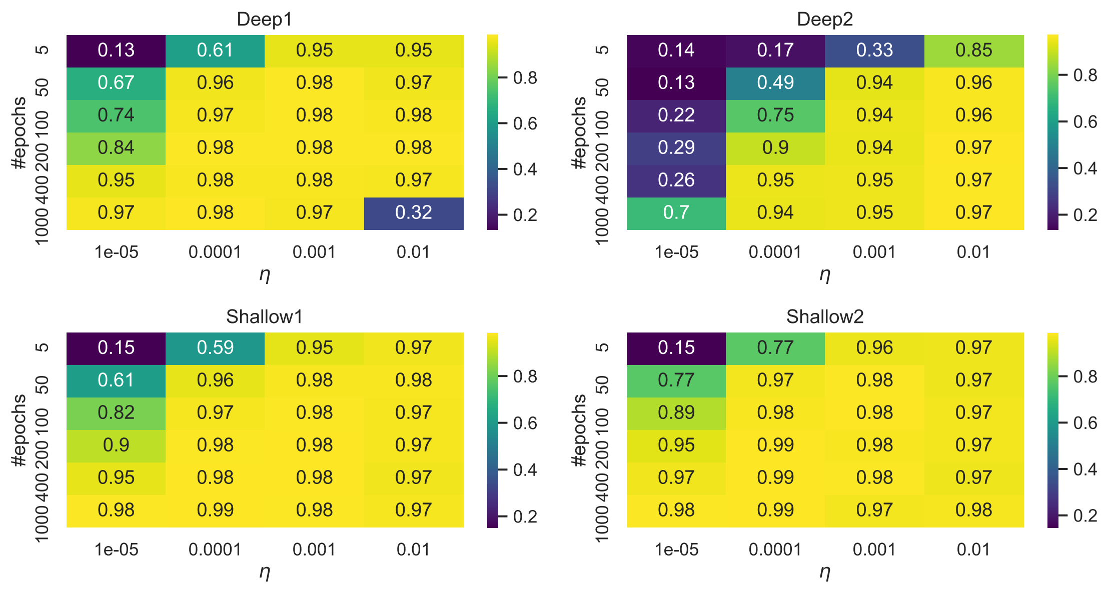

Figure 11: Accuracy on test set for Four different NN- architectures: *Shallow1*, *Shallow2*, *Deep1* and *Deep2*.100 batch size, solver ADAM, Activation function LeakyReLU, L2- regularization 0.0001, 4 folds in K- fold Cross validation.

We see that doubling the number of neurons in the single layer perceptron makes some difference, namely increasing the accuracy overall, and in two instances, bringing it all the way up to 99%. Looking at *Deep1*, we see that adding two more layers of equal thickness to *Shallow1* actually makes the model worse.
It's first in the thin and deep network architecture *Deep2* that we see considerable change. Despite having more neurons than the 100 neuron single layer perceptron, it makes substantially worse predictions. In this case at least, it's clear that little improvement is made in making the single layer perceptron more complicated.

5.3.3 Impact of Hidden Layer Activation Function

Every neuron in the hidden layers of our neural network is equipped with the same activation function. The choice we make ought to influence the numerical stability and convergence of the model. To examine this, we plot the accuracy of the models on test data from 4- fold Cross Validation. Varying over a range of learning rates and number of epochs. The result is shown below in figure 12.

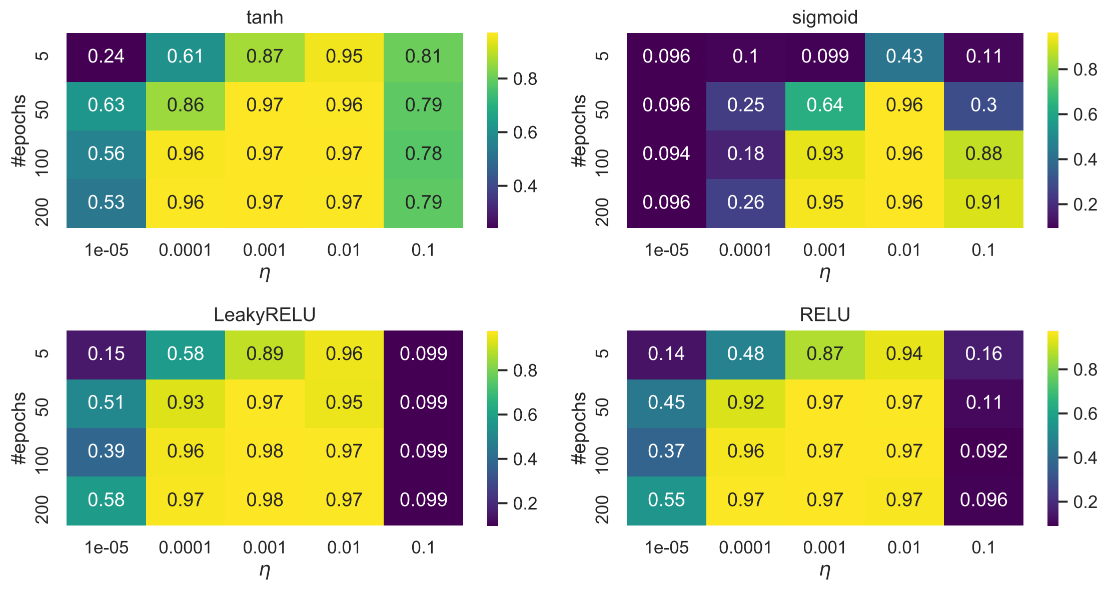

Figure 12: Four different hidden layer activation functions. Number of epochs on the y- axis, learning rate η on the x- axis. Accuracy in proportion of correctly classified digits in test set of each model over 4 folds. Batch size 100, L2 regularization parameter 0.0001, solver = ADAM and architecture [100, 100, 50, 50].

A score close to 0.1 is indicative of a random guess with the most likely candidate for failure being numerical instability or extremely slow convergence. Slow convergence seems to be the case for the sigmoid function which has very poor results for the smallest learning rate. For the next smallest learning rate we see an improvement with a ten fold increase in the number of epochs followed by little improvement with a doubling and quadrupling.
ReLU and LeakyReLU seems to suffer from numerical instability at the largest learning rate with a much more aggressive difference from the next largest learning rate compared to the tanh activation function.
LeakyReLU for this particular configuration of solver, architecture and regularization parameter, gives the best score of 98% for learning rates on the order of 10^{-3} and 200 epochs.

5.4 Multinomial Logistic Regression for Digit Classification

5.4.1 What is wrongly classified?

Our model does make wrong predictions. It's worthwhile to examine what images it classifies wrongly in order to understand where the problem arises. Figure 13 below shows a random sample of four digits that were wrongly classified with their attached labels and the prediction of the model.

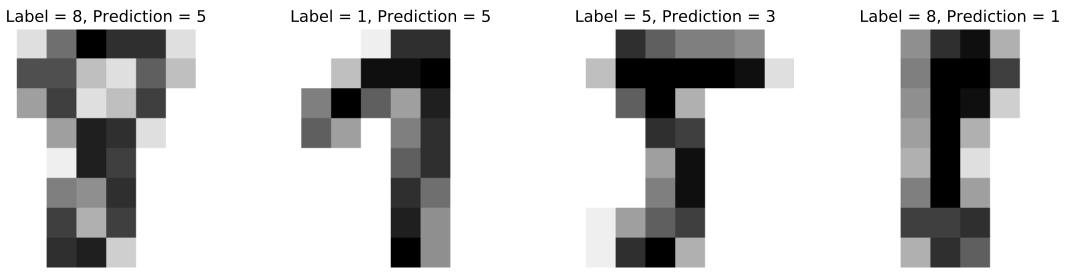
Label = 8, Prediction = 5
Label = 1, Prediction = 5
Label = 5, Prediction = 3
Label = 8, Prediction = 1

Figure 13: Random selection of 4 wrongly classified handwritten digits in the testing set. Classification was made using multinomial logistic regression with stochastic gradient descent. Using 10 epochs and a learning rate of 0.001. Model accuracy was 95.56%.

Arguably, one of the benchmarks by which we should measure the success of our algorithm, is human judgement. Recruiting the aid of an impartial judge, who had not seen either the labels or predictions. They determined from left to right in figure 13, that the digits looked like 9, 1, 5 and 1 respectively. The first and last of which are wrong according to their labels. With this in mind we can conclude that a not insignificant portion of the wrongly classified images lack the pixel resolution needed to meaningfully distinguish them from another digit.

5.4.2 Impact of Learning Rates

We want to examine the role that the learning rate plays in the testing accuracy of our models. Using 150 epochs and a batch size of 1, we get figure 14 below.

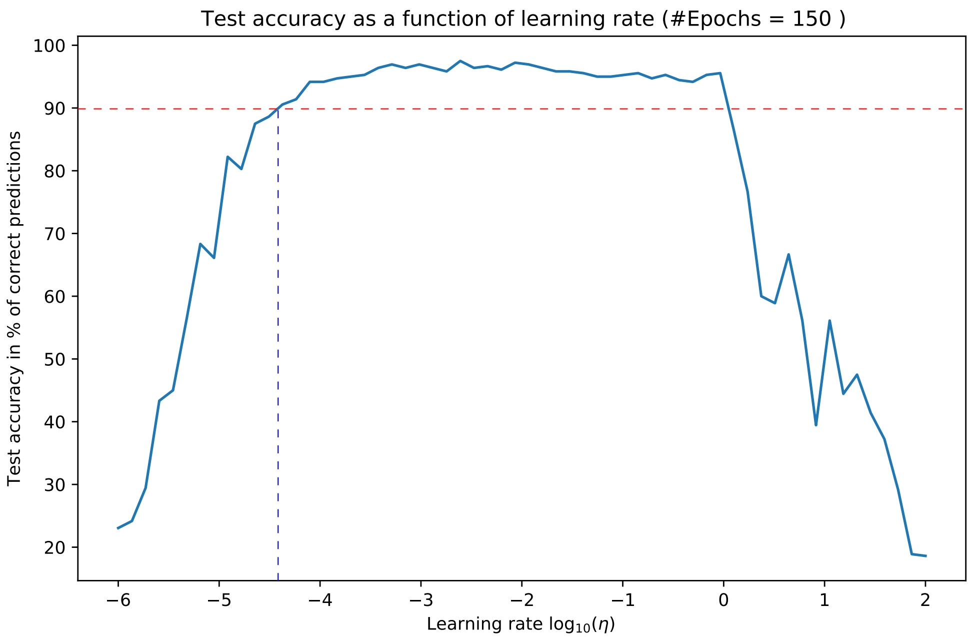

Figure 14: Multinomial logistic regression with stochastic gradient descent. Classification accuracy on the test set as a function of learning rate. Using 150 epochs and a bach size of 1.

We see that there's a stable region between $\eta = 10^{-4}$ and $\eta = 1$ that results in relatively high accuracies. We expect that too large learning rates should prevent the algorithm from converging on an optimum, instead oscillating around it. Too small learning rates may not converge by the time 150 epochs have elapsed. With this in mind, we should expect to see the graph peak at lower learning rates if the algorithm is given more epochs for convergence. We can see this effect in figure 15 below.

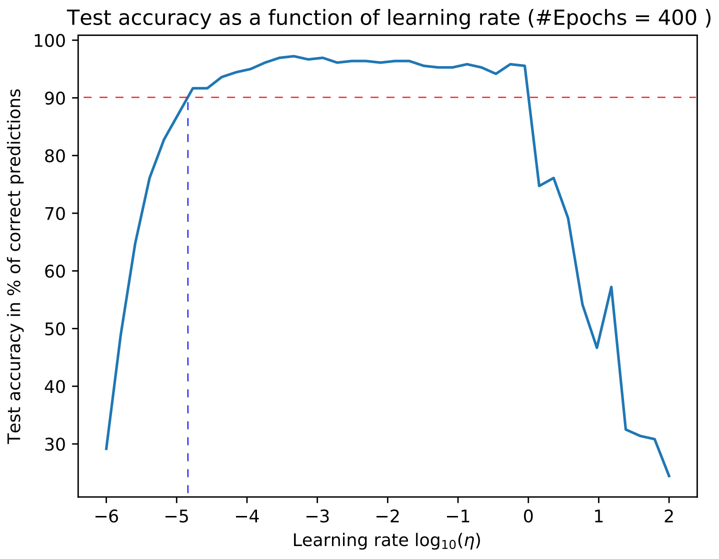

Figure 15: Multinomial logistic regression with stochastic gradient descent. Classification accuracy on the test set as a function of learning rate. Using 400 epochs and a bach size of 1.

Here, the threshold for an accuracy above 90% has shifted towards the smaller learning rate with an increase in the number of epochs.

5.4.3 Comparing Implementation to Scikit- learn

Using a learning rate $\eta = 0.0001$ and 500 epochs on 6 independent runs. Comparing the Scikit- learn implementation with our own implementation of Logistic regression with Stochastic regression we get the results in table 1.

Table 1: Own implementation of Softmax regression with stochastic gradient descent compared to a Scikit- Learn implementation using the LBFGS. Classification accuracy on a test set randomized between each run. Using 500 epochs for both methods and a learning rate of 0.0001 and batch size of 1 for the SGD solver.

Run	Softmax Regression with SGD	Scikit Logistic Regressor with LBFGS
1	97.22%	95.83%
2	95.83%	94.17%
3	97.22%	96.94%
4	96.67%	95.56%
5	94.44%	95.56%
6	96.11%	96.67%

The key difference between these two different implementations is the use of LBFGS solver utilized by the LogisticRegression function (default setting) in Scikit- learn. We can see that their accuracy is roughly the same across runs. This seems to indicate that we have correctly implemented a Softmax regressor using SGD.

6 Conclusion

We have seen that stochastic gradient methods are well suited to approximate the ideal OLS and Ridge parameters, with ADAM being able to achieve a test MSE only 20% higher at 1000 epochs with a batch size of 16 compared to the ideal Ridge parameter. However, the analytical solution is always to be preferred, given it exists.
As can be see in the preceding section on regression, getting results that surpass Ridge Regression and OLS is not difficult with a FFNN, which shows that neural networks are well suited for regression problems. However, finding the ideal parameters is no easy task, as there are a lot of activation functions, gradient methods and parameters to choose. There is also the aspect of time - higher epochs lead to superior results, but the error reduces extremely slowly as the number of epochs grows exponentially [2]. For the regression problem, we found, with help of the sigmoid function, that ADAM worked best as stochastic gradient method, whereas the tanh activation function worked best as activation function (with 2 hidden layers), giving 60% of the test MSE that OLS produces. However, we observed that the Rectifier activation functions, especially LeakyReLU, perform better for deeper neural networks, and we hence suppose that the error can be further reduced by using LeakyReLU as activation function with more epochs. Ultimately however, the Neural Network cannot hide the fact that regression on terrain data cannot yield exact results because of the complicated shape & the fact that sampling with so few points cannot be correct.

In the classification problem we find that the thick and shallow network architectures i.e. *Shallow1* and *Shallow2*, produce the best results over a range of epochs and learning rates, keeping everything else constant. This suggests that the particular problem of classifying an 8×8 grid of pixels, does not benefit from the addition of more hidden layers with the number of epochs we're considering.
We found that, within the range of epochs we looked at and the particular choice of architecture, LeakyReLU produced models with the highest accuracy of 99%, with both tanh and ReLU following closely behind with consistent accuracies of 97%. Sigmoid, at least for this number of epochs produced considerably worse results.
In Multinomial logistic regression, we looked at some of the wrongly classified digits and found at least some of them to be indistinguishable from another digit, suggesting that the limited pixel value of the dataset is reducing the feasible accuracy of our models. Looking at how we might interpret accuracy scores when tweaking learning rate and the number of epochs in our parameter space, we found (unsurprisingly) that allowing for more epochs makes lower learning rates perform better.
As for whether logistic regression or neural networks are better for digit classification, we saw clearly better results when using neural networks, although the time required for convergence was consistently much higher for neural networks.
One way to improve on this is report is to implement proper cutoff procedures for numerical instability to reduce run time, since bad parameters give overflow errors and hence unusable results. This includes stopping after the test accuracy stops improving or when the train accuracy is 1. It is absolutely possible to improve this implementation's run time by being more strict in using vectorized code and avoiding unnecessary memory allocations by doing in-place manipulation of data (or writing the code in a compiled language such as C). Different type of Neural Networks give much improved results for classification. With a deep FFNN and deformed training images, it is possible to get the test error to 0.35% and below [1]. Using Convolutional Neural Networks, it is possible to get the error down to 0.21% [9], and it looks like there is still room for improvement. We are quite convinced that we did not find optimal parameters, as we did not explore the full parameter space of these methods. Our search search was mainly restricted to a simple grid search and a limited number of epochs and hidden layers.

We also did not implement the full MNIST dataset, but only a reduced version thereof, and it would be interesting to see how well our simple network performs on the full data, both in terms of speed and quality.

7 Appendix

7.1 Proof that Softmax reduces to the Logistic function for m=2

Consider the Softmax function for $m = 2$ categories

$$\mathbf{a} = \frac{1}{\exp(\mathbf{W}_{[1,:]} \cdot \mathbf{x}^T) + \exp(\mathbf{W}_{[2,:]} \cdot \mathbf{x}^T)} \begin{pmatrix} \exp(\mathbf{W}_{[1,:]} \cdot \mathbf{x}^T) \\ \exp(\mathbf{W}_{[2,:]} \cdot \mathbf{x}^T) \end{pmatrix} \tag{22}$$

We can multiply both sides of the fraction by $\exp(-\mathbf{W}_{[2,:]} \cdot \mathbf{x}^T)$ to get

$$\mathbf{a} = \frac{1}{1 + \exp((\mathbf{W}_{[1,:]} - \mathbf{W}_{[2,:]}) \cdot \mathbf{x}^T)} \begin{pmatrix} \exp((\mathbf{W}_{[1,:]} - \mathbf{W}_{[2,:]}) \cdot \mathbf{x}^T) \\ \exp(\vec{0}) \end{pmatrix} \tag{23}$$

We can now simply rename this weight parameter as $\mathbf{W}_{[1,:]} - \mathbf{W}_{[2,:]} = -\mathbf{W}$ giving us

$$a = \begin{pmatrix} \frac{\exp(-\mathbf{W} \cdot \mathbf{x}^T)}{1+\exp(-\mathbf{W} \cdot \mathbf{x}^T)} \\ \frac{1}{1+\exp(-\mathbf{W} \cdot \mathbf{x}^T)} \end{pmatrix} = \begin{pmatrix} 1 - \frac{1}{1+\exp(-\mathbf{W} \cdot \mathbf{x}^T)} \\ \frac{1}{1+\exp(-\mathbf{W} \cdot \mathbf{x}^T)} \end{pmatrix} \tag{24}$$

which is the same as the activation function for logistic regression.

7.2 Figures

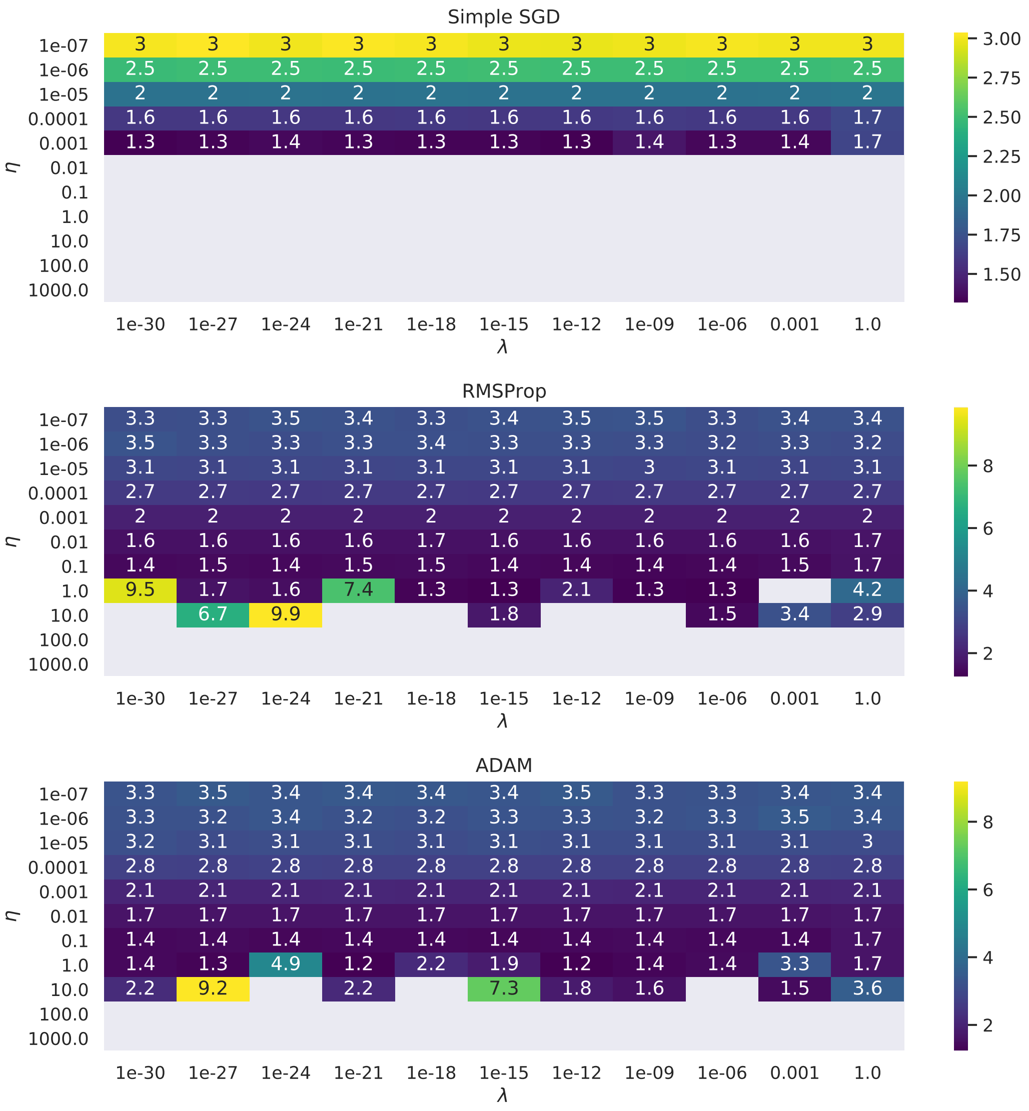

Figure 16: Relative Test MSE ($\frac{MSE_{SGD}}{MSE_{analytical}}$) for the simple SGD method, RMSProp and ADAM and as functions of the learning rate η and the regularization parameter λ. $N = 2000$, the polynomial degree used is $deg = 20$, where OLS fails. The batch size is 16, the number of epochs is 1000. Values exceeding 10 were removed, explaning the grey parts. No bootstrapping or cross-validation was performed.

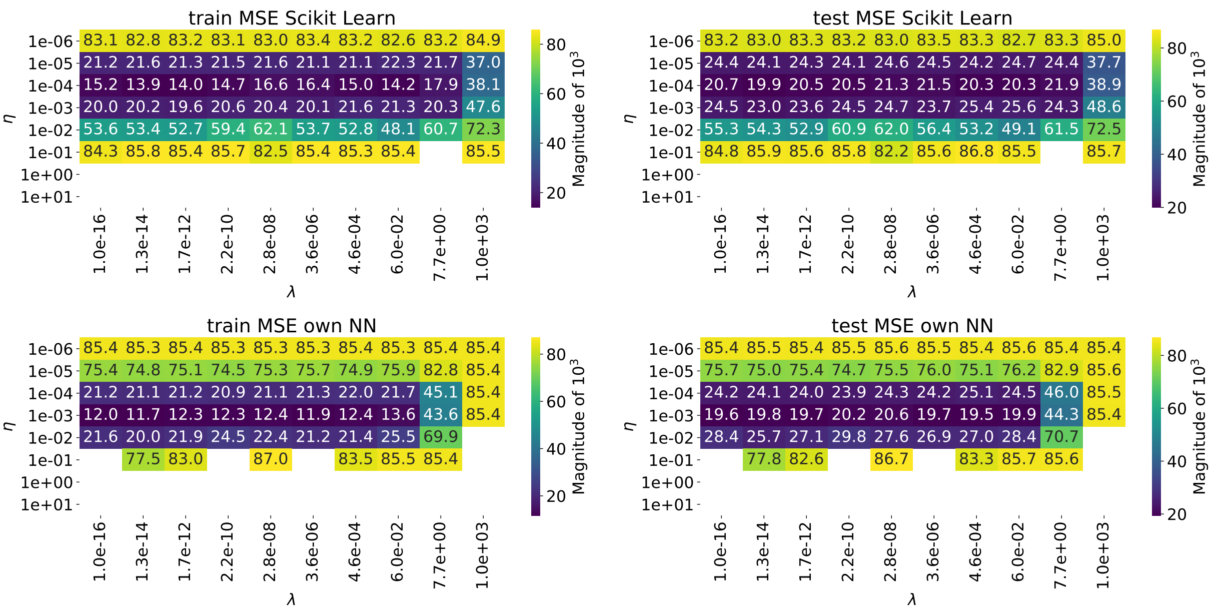

Figure 17: Train and test MSE <u>divided by 1000</u> as function of the learning rate η and the regularization parameter λ for our own Neural Network and Scikit-learn 's MLPRegressor function. We used 2000 data points (randomly selected), a polynomial degree of 20, a batch size of 200, 1000 epochs, two hidden layers with 100 neurons each, simple SGD as gradient descent method and the sigmoid function as activation function between the layers. Values exceeding 100,000 are excluded from the plot. We used 5-fold Cross Validation to estimate the errors.

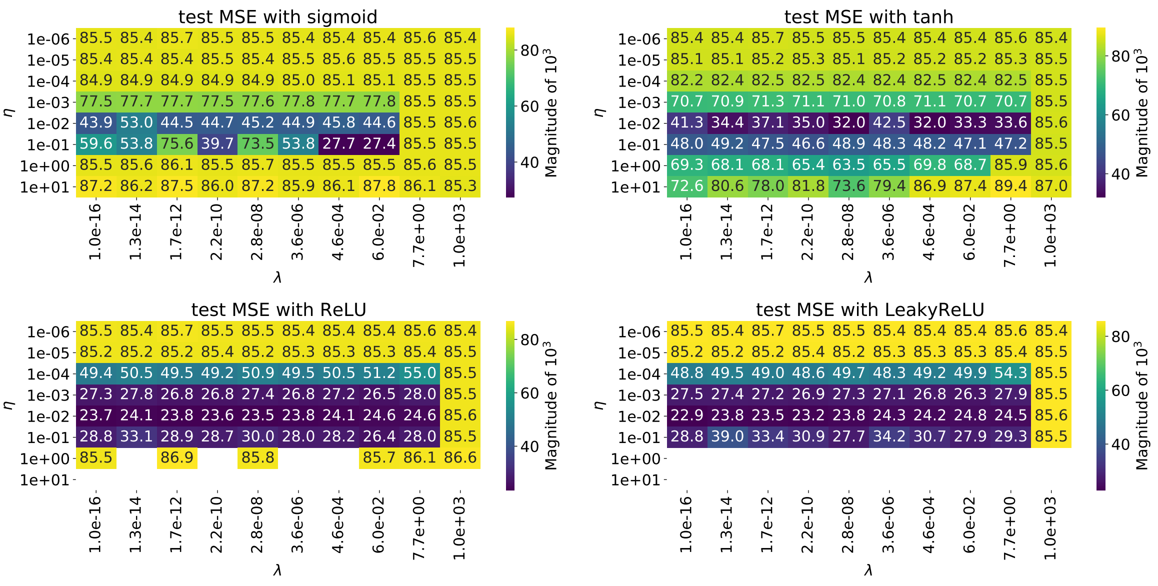

Figure 18: Test MSE <u>divided by 1000</u> as function of the learning rate η and the regularization parameter λ for our own Neural Network using ADAM. We used 2000 data points (randomly selected), a polynomial degree of 10, a batch size of 200, 1000 epochs, 4 hidden layers with respectively 100, 100, 50 and 50 neurons and the activation function stated in the title between the layers. Values exceeding 100,000 are excluded from the plot. We used 5-fold Cross Validation to estimate the errors.

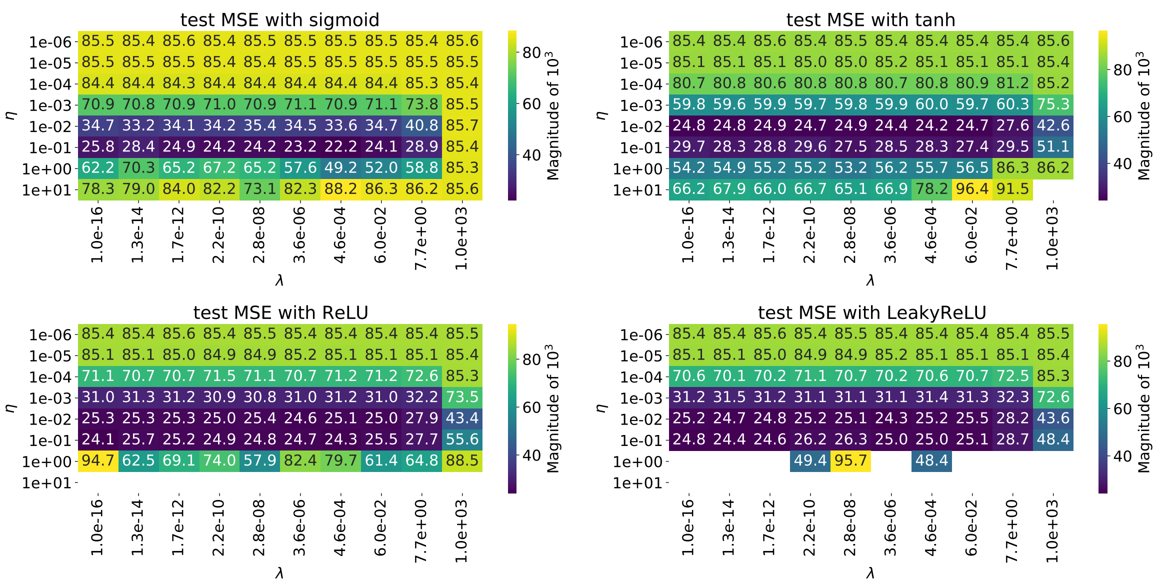
Figure 19: Test MSE divided by 1000 as function of the learning rate η and the regularization parameter λ for our own Neural Network using ADAM. We used 2000 data points (randomly selected), a polynomial degree of 10, a batch size of 200, 100 epochs, two hidden layers with 100 neurons each and the activation function stated in the title between the layers. Values exceeding 100,000 are excluded from the plot. We used 5-fold Cross Validation to estimate the errors.

References

[1] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, and Juergen Schmidhuber. Deep big simple neural nets excel on handwritten digit recognition. 2010.

[2] Aurelien Geron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* O'Reilly Media, Inc., 1st edition, 2017.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, page 1026–1034, USA, 2015. IEEE Computer Society.

[4] Morten Hjorth-Jensen. Data analysis and machine learning: Lecture notes. 2020.

[5] A. Kleven and S. Schrader. Project 1 in fys-stk4155. 2020.

[6] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[7] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[9] V.V. Romanuke. Training data expansion and boosting of convolutional neural networks for reducing the MNIST dataset error rate. *Research Bulletin of the National Technical University of Ukraine "Kyiv Polytechnic Institute"*, 0(6):29–34, December 2016.

[10] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stefan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, lhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antonio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. Scipy 1.0: Fundamental algorithms for scientific computing in python. *Nature Methods*, 2020.