Contents

List of Figures

List of Tables

1 Abstract

Machine learning - is it possible to learn this power? - Anakin
Strong in you The computational power is! - Yoda
It doesn't converge! This is outrageous! This is unfair! How can you call something machine learning but don't learn nothing at all? - Anakin
Meesa computer master now! - Jar Jar Binks
Hello np.where()! - Obi Wan Kenobi
INFINITE CONVERGENCE - The senate

2 Introduction

As can be seen in [**?**], both Ordinary Least Square (OLS) regression and Ridge regression failed to accurately fit a polynomial function to geographic 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 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 neutrons per hidden layer.
We also tested the neural network's performance on a categorization problem, namely the MNIST data set [**?**]. We compared several activation functions for the hidden layers, while using the Softmax function (or the sigmoid function) for the output layer. Finally, we compared the neural network's performance to the results achieved using logistic regression.
In this report, we will first describe the methods used, focusing on stochastic gradient descent methods and the theory of FFNNs. This is done in the method part. The result part cointains

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.

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 1 below, the functions and variables in the nodes are as given in equations (1) to (5).

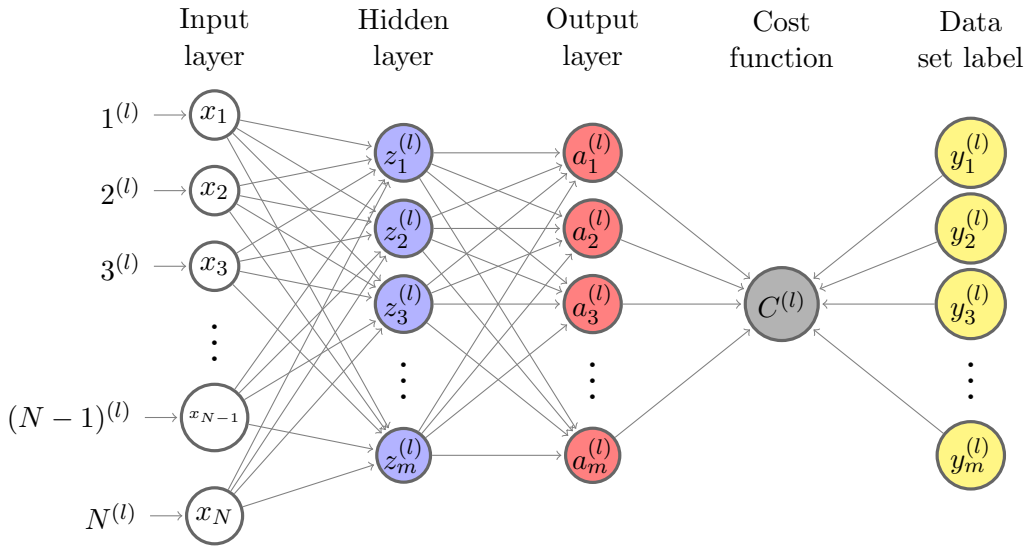

Figure 1: The single layer perceptron model of multinomial logistic regression.

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 Back propagation

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:

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 [**?**]. 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

$$\nabla_\beta C(\beta) = \frac{2}{m}\left[X^T\left(X\beta - y\right) + \lambda\beta\right] \tag{9}$$

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:

$$\beta^{i+1} = \beta^i - \gamma \nabla_\beta C(\beta^i) \tag{10}$$

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 [**?**]. 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 (10) 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

$$\gamma_t = \frac{t_0}{t_1 + t} \tag{11}$$

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 (10), 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

$$\begin{aligned} \boldsymbol{s}^{i+1} &= \alpha \boldsymbol{s}^i + (1-\alpha)\nabla_\beta C(\boldsymbol{\beta^i}) * \nabla_\beta C(\boldsymbol{\beta^i}) \\ \boldsymbol{\beta}^{i+1} &= \boldsymbol{\beta}^i - \gamma C(\boldsymbol{\beta^i}) / \sqrt{\boldsymbol{s}^{i+1} + e} \end{aligned} \tag{12}$$

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 [**?**]) - 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

$$\begin{aligned} \boldsymbol{m}^{i+1} &= [\alpha_1 \boldsymbol{m} + (1-\alpha_1)\nabla_\beta C(\boldsymbol{\beta^i})](1-\alpha_1^T)^{-1} \\ \boldsymbol{s}^{i+1} &= \alpha_2 \boldsymbol{s}^i + (1-\alpha_2)\nabla_\beta C(\boldsymbol{\beta^i}) * \nabla_\beta C(\boldsymbol{\beta^i})(1-\alpha_2^T)^{-1} \\ \boldsymbol{\beta}^{i+1} &= \boldsymbol{\beta}^i - \gamma \boldsymbol{m}^{i+1} / \sqrt{\boldsymbol{s}^{i+1} + e} \end{aligned} \tag{13}$$

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

3.3.1 Feed forward?

3.3.2 Back propagation

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

sigmoid function - The sigmoid function, defined as $\sigma(x) = \frac{e^x}{e^x - 1}$, can take functions between 0 and 1. It is inspired by the way Neurons fire in the brain [XXX].

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.

3.4 Data sets

3.4.1 Regression: Geographic Data

For the regression analysis, we used the same data as in [**?**] - 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 [**?**]. 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 Neural network

4.1.1 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 [**?**], 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.

4.1.2 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.

5 Results

5.1 Comparison of SGD methods for OLS

Figure 2 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.

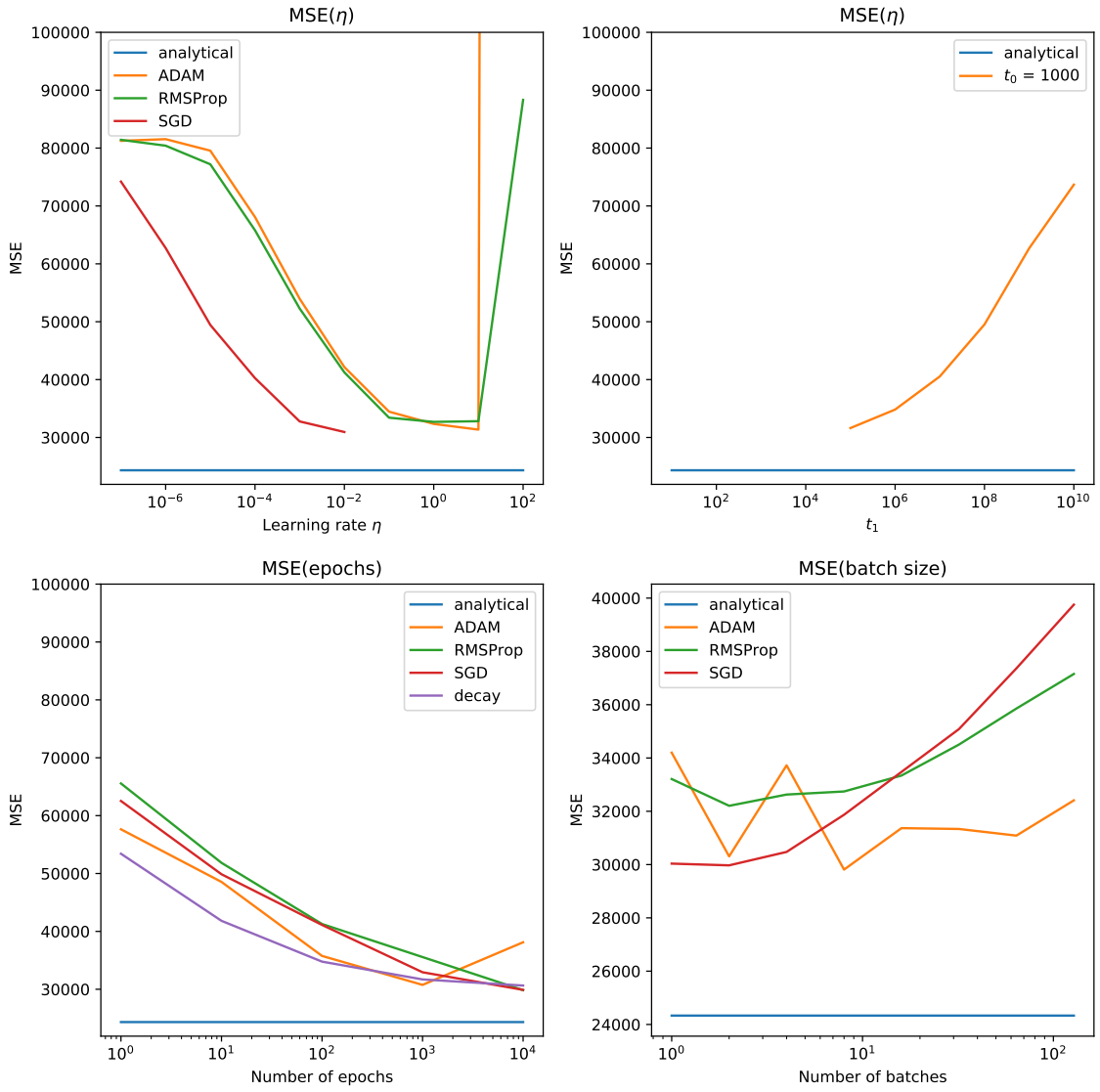

Figure 2: The simple SGD method (titled SGD), RMSProp, ADAM and decaying η as functions of the learning rate η (top left), the number t_1 (top right), the number of epochs (down left), and different batch sizes (down right). The number of data points is $N = 2000$, the polynomial degree used is $deg = 10$. 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 3 contains the same plots as figure 2, 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.

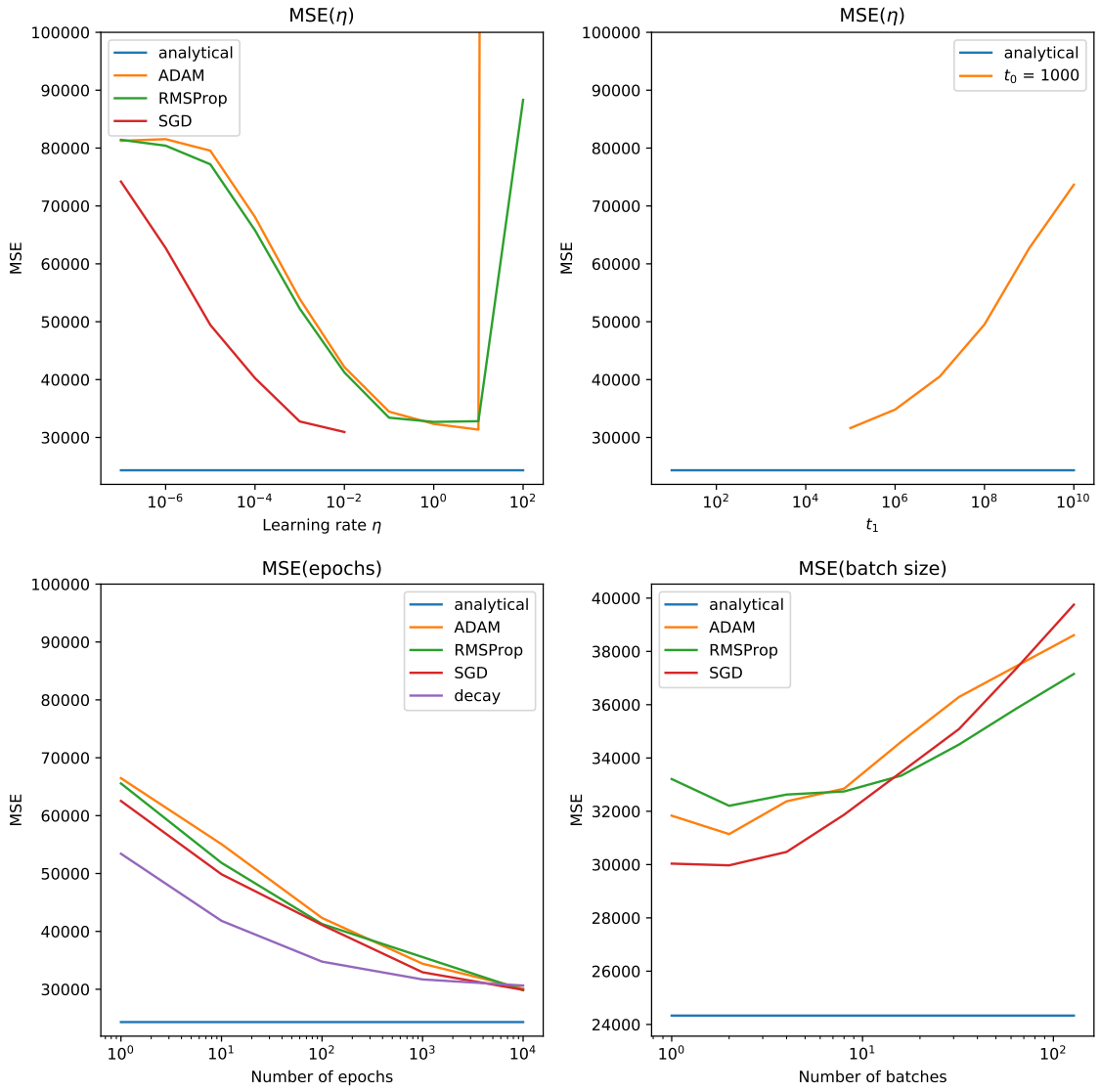

Figure 3: The simple SGD method (titled SGD), RMSProp, ADAM and decaying η as functions of the learning rate η (top left), the number t_1 (top right), the number of epochs (down left), and different batch sizes (down right). The number of data points is $N = 2000$, the polynomial degree used is $deg = 10$. 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 8 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 geographic data [**?**] 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 4 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 [**?**], which are included in figure 4.

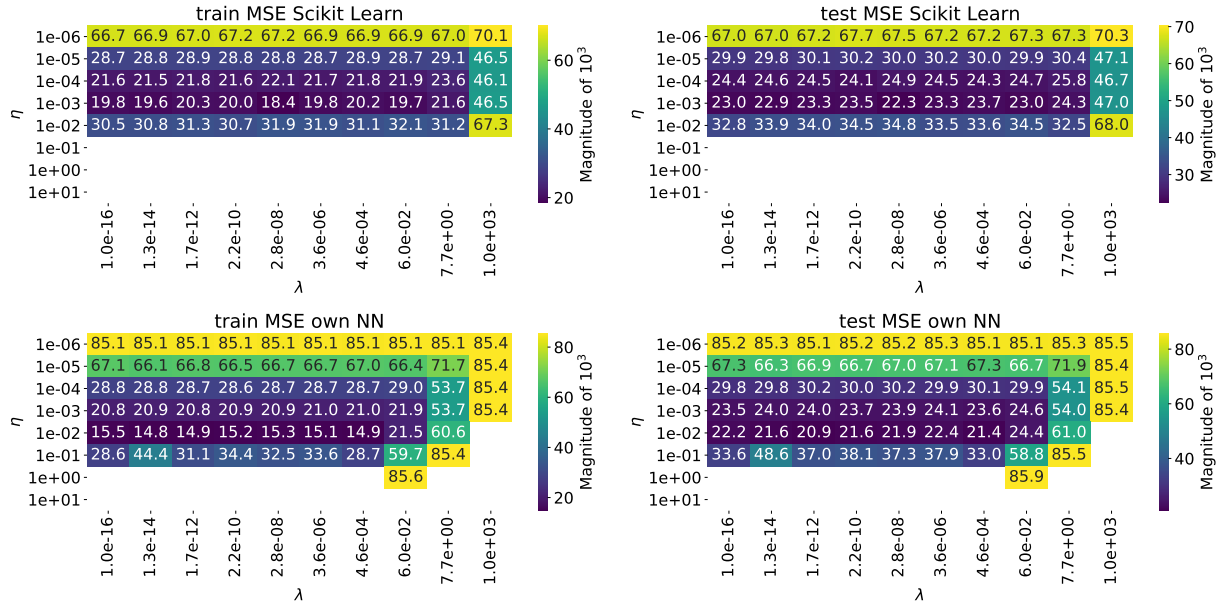

Figure 4: Train and test MSE divided by 1000 as function of the learning rate η and the regularization parameter λ for our own Neural Network and Scikit-learn 's MLPRegressor function. We used 2000 data points (randomly selected), a polynomial degree of 10, a batch size of 200, 1000 epochs, one hidden 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 [**?**], 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, as is clear as 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 4) was done, this time using two hidden layers with 100 neurons each. The results can be seen in figure 5.

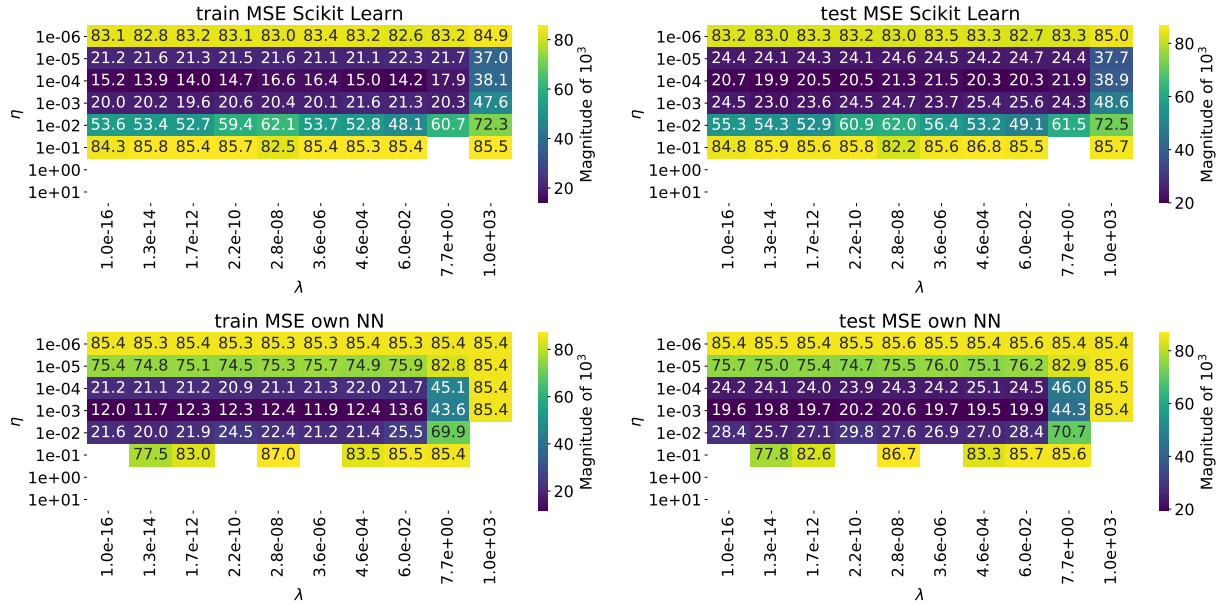

Figure 5: Train and test MSE divided by 1000 as function of the learning rate η and the regularization parameter λ for our own Neural Network and Scikit-learn 's MLPRegressor function. We used 2000 data points (randomly selected), a polynomial degree of 10, a batch size of 200, 1000 epochs, two hidden 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 9 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 6 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 5, that is, two hidden layers with 100 neurons each.

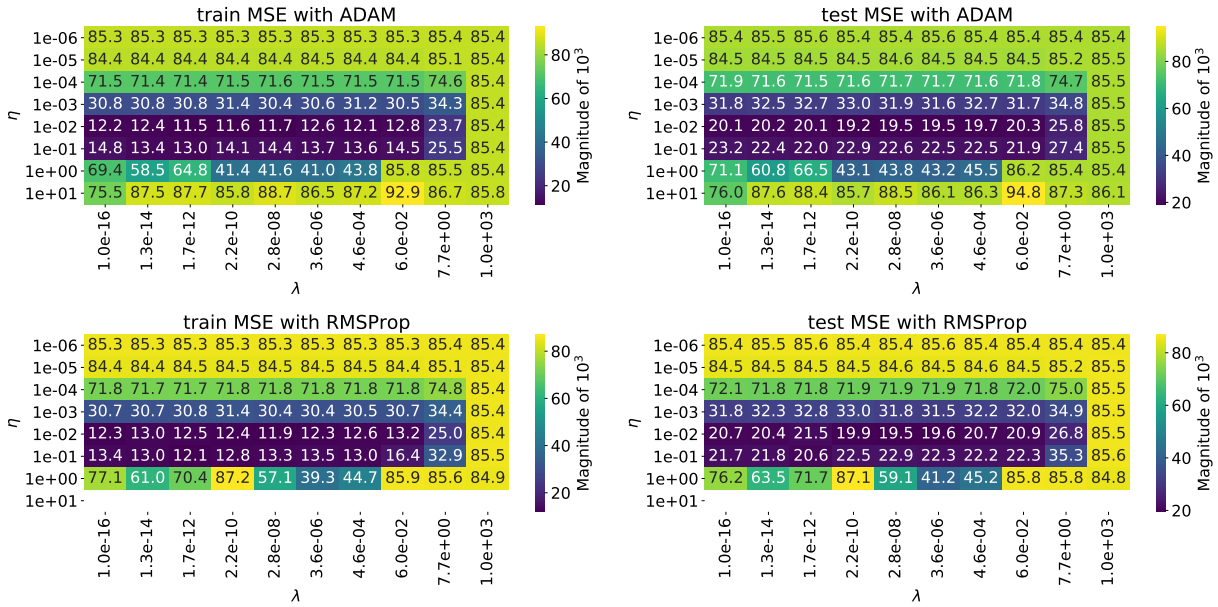

Figure 6: Train and test MSE divided by 1000 as function of the learning rate η and the regularization parameter λ for our own Neural Network using ADAM and RMSProp. We used 2000 data points (randomly selected), a polynomial degree of 10, a batch size of 200, 1000 epochs, two hidden layers with 100 neurons 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 7.

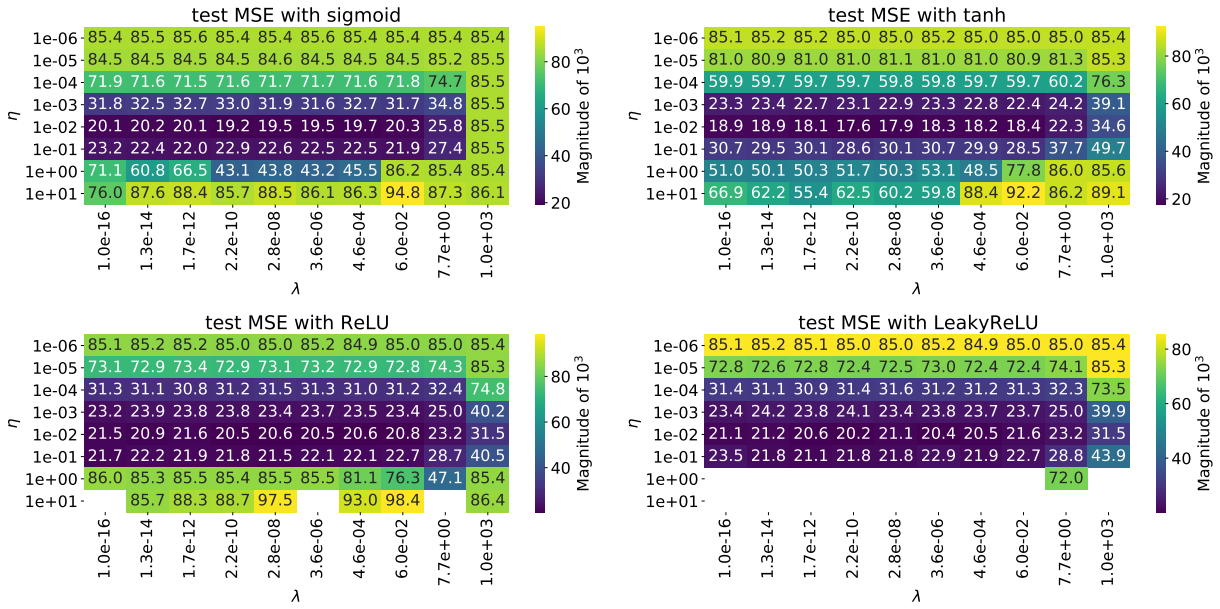

Figure 7: Test MSE divided by 1000 as function of the learning rate η and the regularization parameter λ for our own Neural Network using ADAM. We used 2000 data points (randomly selected), a polynomial degree of 10, a batch size of 200, 1000 epochs, two hidden layers with 100 neurons each and the activation 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 10 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 11 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.

6 Conclusion

As can be see in the preceding section on regression, getting results that surpass Ridge Regression and OLS is not difficult, 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 slow as the number of epochs grows exponentially [**?**]. 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, as the network got deeper, we found that...XXX blablabla

7 Appendix

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

Consider the Softmax function for $m = 2$ categories

$$\mathbf{a} = \frac{1}{\exp(\mathbf{W}_{[1,:]} \cdot \mathbf{x}^T) + \exp(\mathbf{W}_{[2,:]} \cdot \mathbf{x}^T)} \begin{pmatrix} \exp(\mathbf{W}_{[1,:]} \cdot \mathbf{x}^T) \\ \exp(\mathbf{W}_{[2,:]} \cdot \mathbf{x}^T) \end{pmatrix} \tag{14}$$

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

$$\mathbf{a} = \frac{1}{1 + \exp((\mathbf{W}_{[1,:]} - \mathbf{W}_{[2,:]}) \cdot \mathbf{x}^T)} \begin{pmatrix} \exp((\mathbf{W}_{[1,:]} - \mathbf{W}_{[2,:]}) \cdot \mathbf{x}^T) \\ \exp(\vec{0}) \end{pmatrix} \tag{15}$$

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

$$a = \begin{pmatrix} \frac{\exp(-\mathbf{W} \cdot \mathbf{x}^T)}{1+\exp(-\mathbf{W} \cdot \mathbf{x}^T)} \\ \frac{1}{1+\exp(-\mathbf{W} \cdot \mathbf{x}^T)} \end{pmatrix} = \begin{pmatrix} 1 - \frac{1}{1+\exp(-\mathbf{W} \cdot \mathbf{x}^T)} \\ \frac{1}{1+\exp(-\mathbf{W} \cdot \mathbf{x}^T)} \end{pmatrix} \tag{16}$$

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

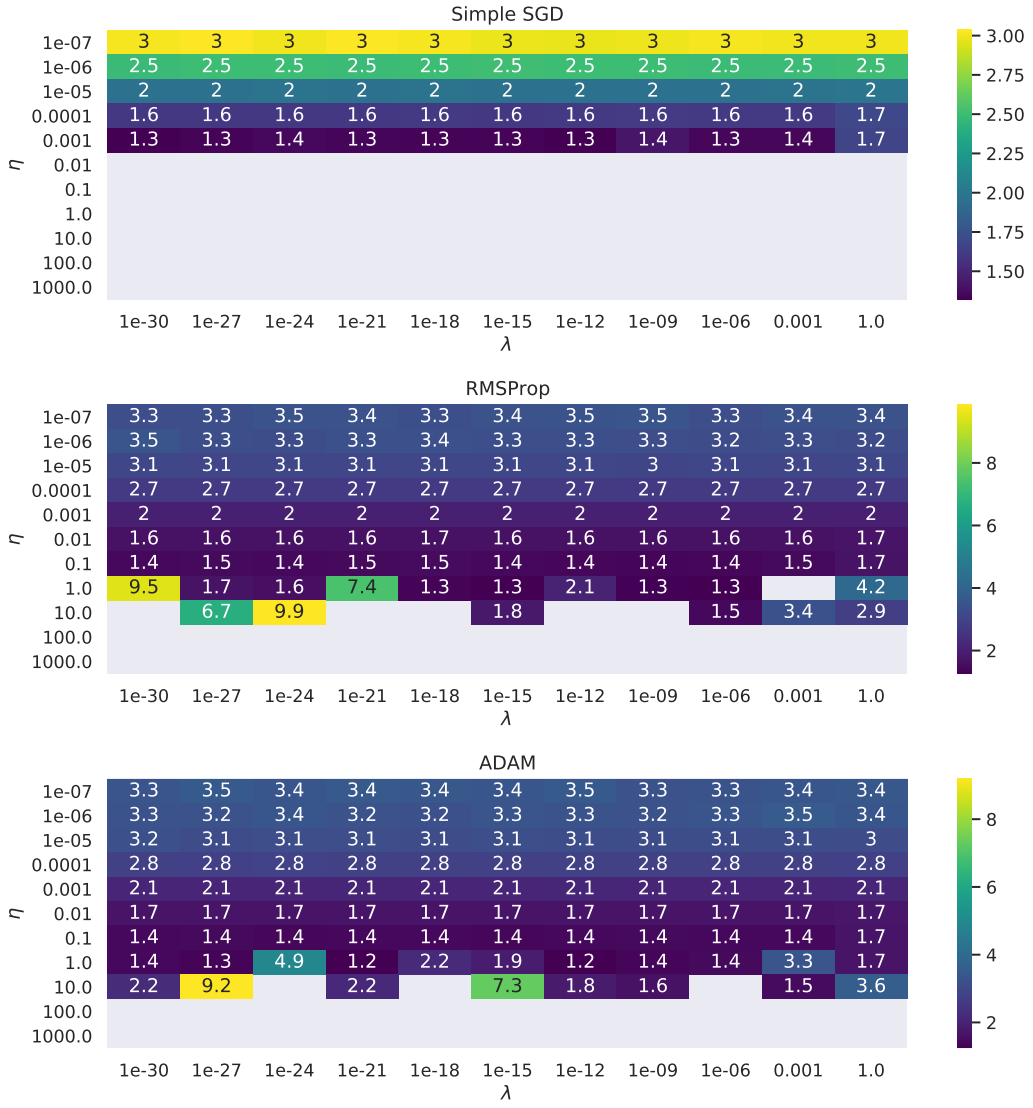

Figure 8: Relative Test MSE ($\frac{MSE_{SGD}}{MSE_{analytical}}$) for the simple SGD method, RMSProp and ADAM and as functions of the learning rate η and the regularization parameter λ. $N = 2000$, the polynomial degree used is $deg = 20$, where OLS fails. The batch size is 16, the number of epochs is 1000. Values exceeding 10 were removed, explaning the grey parts. No bootstrapping or cross-validation was performed.

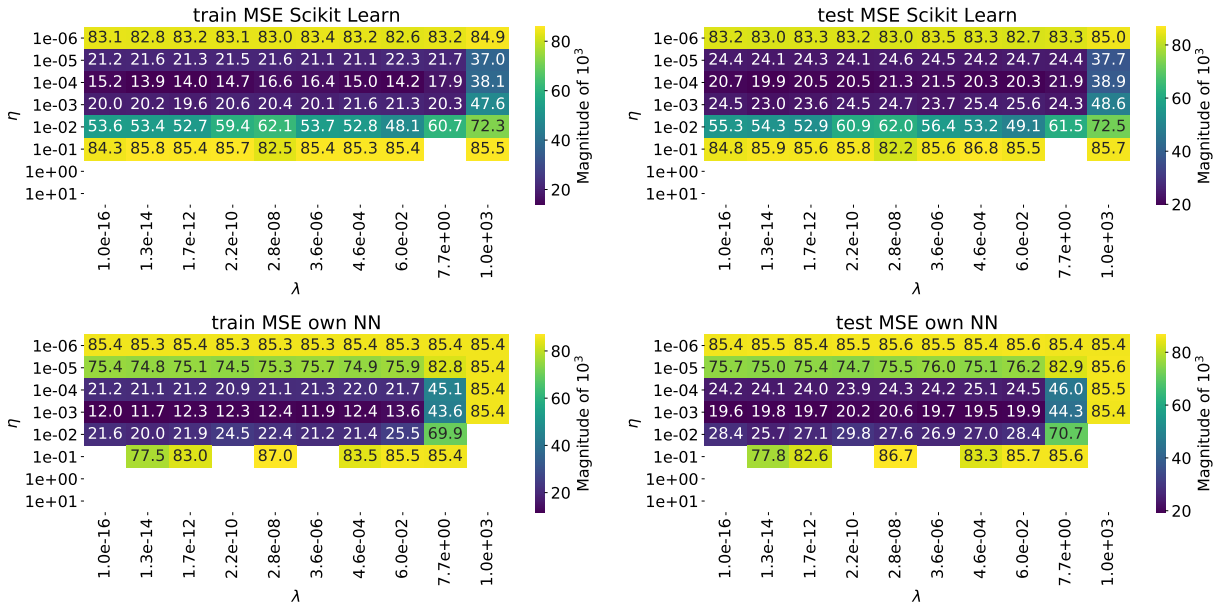

Figure 9: Train and test MSE divided by 1000 as function of the learning rate η and the regularization parameter λ for our own Neural Network and Scikit-learn 's MLPRegressor function. We used 2000 data points (randomly selected), a polynomial degree of 20, a batch size of 200, 1000 epochs, two hidden 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.

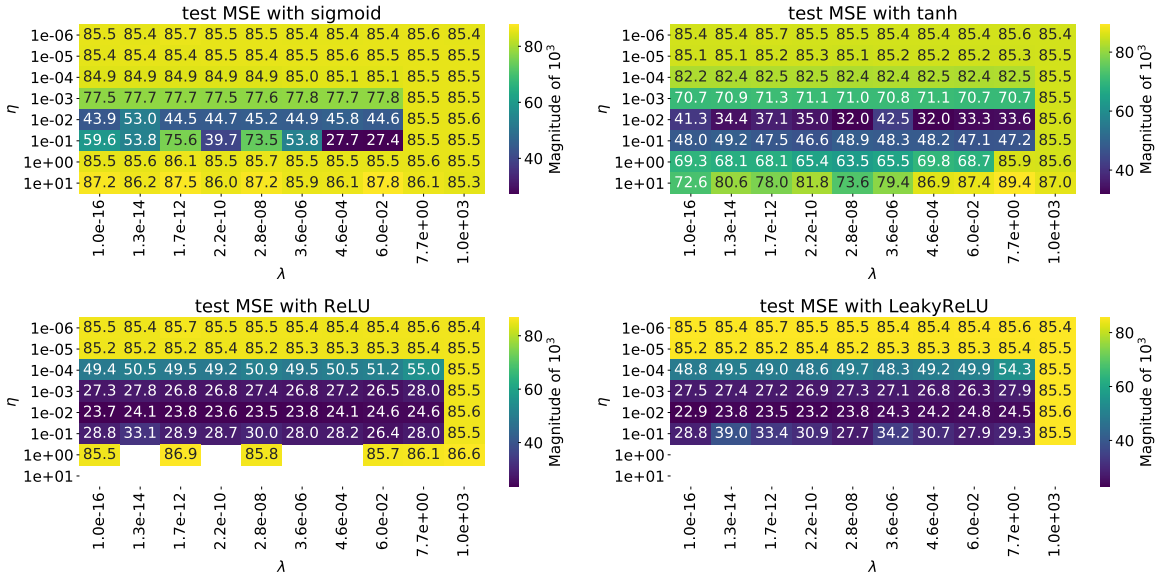

Figure 10: Test MSE divided by 1000 as function of the learning rate η and the regularization parameter λ for our own Neural Network using ADAM. We used 2000 data points (randomly selected), a polynomial degree of 10, a batch size of 200, 1000 epochs, 4 hidden layers with respectively 100, 100, 50 and 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.

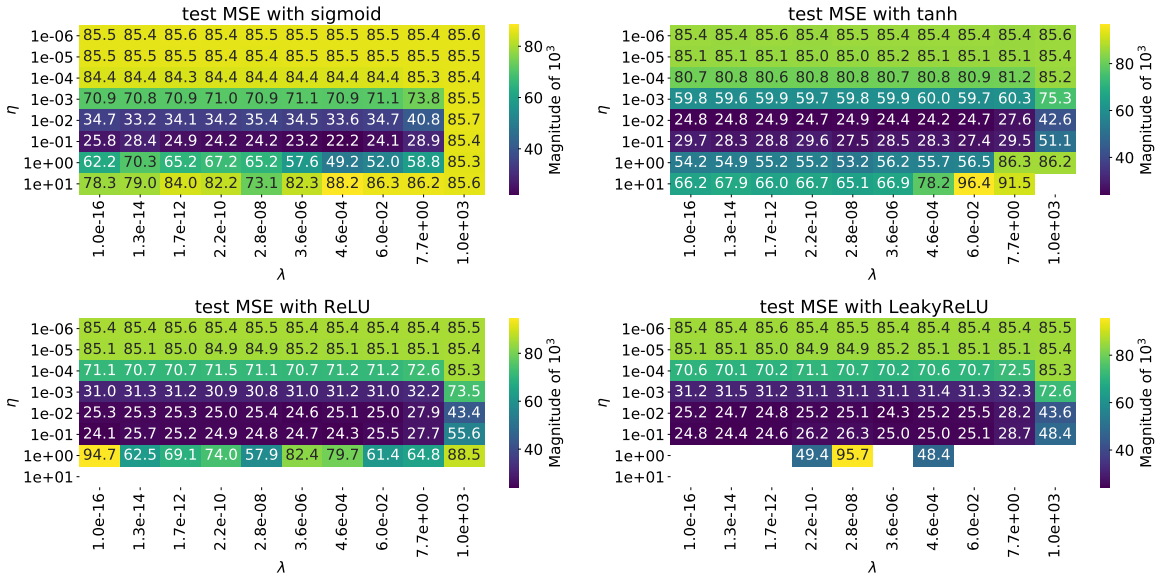

Figure 11: Test MSE divided by 1000 as function of the learning rate η and the regularization parameter λ for our own Neural Network using ADAM. We used 2000 data points (randomly selected), a polynomial degree of 10, a batch size of 200, 100 epochs, two hidden layers with 100 neurons each and the activation 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.

7.2 Figures